Pedro R. D'Argenio
Hernán Melgratti (Eds.)

ARCoSS
LNCS 8052

CONCUR 2013 – Concurrency Theory

24th International Conference, CONCUR 2013
Buenos Aires, Argentina, August 2013
Proceedings

Springer

Lecture Notes in Computer Science 8052

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Advanced Research in Computing and Software Science
Subline of Lectures Notes in Computer Science

Pedro R. D'Argenio Hernán Melgratti (Eds.)

CONCUR 2013 – Concurrency Theory

24th International Conference, CONCUR 2013
Buenos Aires, Argentina, August 27-30, 2013
Proceedings

Springer

Volume Editors

Pedro R. D'Argenio
Universidad Nacional de Córdoba – CONICET
Facultad de Matemáticas, Astronomía y Física
Medina Allende s/n
X5000HUA Córdoba, Argentina
E-mail: dargenio@famaf.unc.edu.ar

Hernán Melgratti
Universidad de Buenos Aires - CONICET
Departamento de Computación
Intendente Güirales 2160, Pabellón 1, Ciudad Universitaria
C1428EGA Ciudad Autónoma de Buenos Aires, Argentina
E-mail: hmelgra@dc.uba.ar

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-40183-1 e-ISBN 978-3-642-40184-8
DOI 10.1007/978-3-642-40184-8
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013944556

CR Subject Classification (1998): F.3, D.2, F.1, D.3, F.4, G.3, C.2, H.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 24th Conference on Concurrency Theory (CONCUR 2013) held in Buenos Aires, Argentina, during August 27–30, 2013. CONCUR 2013 was organized by the Universidad de Buenos Aires and the Universidad Nacional de Córdoba.

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. The principal topics include *basic models of concurrency* such as abstract machines, domain theoretic models, game theoretic models, process algebras, and Petri nets; *logics for concurrency* such as modal logics, probabilistic and stochastic logics, temporal logics, and resource logics; *models of specialized systems* such as biology-inspired systems, circuits, hybrid systems, mobile and collaborative systems, multi-core processors, probabilistic systems, real-time systems, service-oriented computing, and synchronous systems; *verification and analysis techniques for concurrent systems* such as abstract interpretation, atomicity checking, model checking, race detection, pre-order and equivalence checking, run-time verification, state-space exploration, static analysis, synthesis, testing, theorem proving, and type systems; *related programming models* such as distributed, component-based, object-oriented, and Web services.

This edition of the conference attracted 115 submissions. We would like to thank all their authors for their interest in CONCUR 2013. After careful reviewing and discussions, the Program Committee selected 34 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 the additional reviewers 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 Lorenzo Alvisi (joint invited speaker with QEST 2013), Joost-Pieter Katoen, Philippe Schnoebelen, and Reinhard Wilhelm (joint invited speaker with FORMATS 2013).

This year the conference was jointly organized with the 10th International Conference on Quantitative Evaluation of Systems (QEST 2013), the 11th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2013), and the 8th International Symposium on Trustworthy Global Computing (TGC 2013).

In addition, CONCUR 2013 included six satellite events:

- Combined 20th International Workshop on Expressiveness in Concurrency and 10th Workshop on Structural Operational Semantics (EXPRESS/SOS 2013), organized by Bas Luttik and Johannes Borgström

- 9th International Workshop on Developments in Computational Models (DCM 2013), organized by Mauricio Ayala-Rincón, Eduardo Bonelli, and Ian Mackie
- Latin American Workshop on Formal Methods (LAFM 2013), organized by Leila Ribeiro and Nazareno Aguirre
- IFIP WG 1.8 Workshop on Trends in Concurrency Theory (TRENDS 2013), organized by Bas Luttik and Jos Baeten
- Young Researchers Workshop on Concurrency Theory (YR-CONCUR 2013), organized by Nicolás D'Ippolito
- MEALS Momentum Gathering, organized by Marcelo Frias

We would like to thank everybody who contributed to the organization of CONCUR 2013, especially the Workshop Organization Chairs Eduardo Bonelli and Diego Garbervetsky, the Proceedings Chair Nicolás Wolovick, the Publicity Chair Damián Barsotti, as well as the Organizing Committee, including Renata D'Amore, Daniela Bonomo, Silvia Pelozo, and Matías D. Lee. We also thank the Facultad de Ciencias Económicas of the Universidad de Buenos Aires for providing the venue location. Furthermore, we gratefully acknowledge the financial support of the Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), the Agencia Nacional de Promoción Científica y Tecnológica (through the RC program of FONCYT and FONSOFT), and the EU FP7 grant agreement 295261 MEALS (Mobility between Europe and Argentina applying Logics to Systems).

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.

August 2013

Pedro R. D'Argenio
Hernán Melgratti

Organization

Steering Committee

Roberto Amadio	Université Paris Diderot, France
Jos Baeten	CWI, The Netherlands
Eike Best	Universität Oldenburg, Germany
Kim G. 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
Eike Best	Universität Oldenburg, Germany
Patricia Bouyer	LSV, CNRS, and ENS Cachan, France
Tomas Brazdil	Masaryk University, Czech Republic
Krishnendu Chatterjee	Institute of Science and Technology, Austria
Rance Cleaveland	University of Maryland, USA
Pedro R. D'Argenio	Universidad Nacional de Córdoba, CONICET, Argentina
Wan Fokkink	Vrije Universiteit Amsterdam, The Netherlands
Daniele Gorla	University of Rome "La Sapienza", Italy
Holger Hermanns	Saarland University, Germany
Radha Jagadeesan	DePaul University, USA
Bengt Jonsson	Uppsala University, Sweden
Kim G. Larsen	Aalborg University, Denmark
Hernán Melgratti	Universidad de Buenos Aires, CONICET, Argentina
Ugo Montanari	Università di Pisa, Italy
Prakash Panangaden	McGill University, Canada
David Parker	University of Birmingham, UK
Frank Pfenning	Carnegie Mellon University, USA
Nir Piterman	University of Leicester, UK
Shaz Qadeer	Microsoft Research, USA
Jean-Francois Raskin	Université Libre de Bruxelles, Belgium
Jan Rutten	CWI, The Netherlands
Davide Sangiorgi	University of Bologna, Italy
Geoffrey Smith	Florida International University, USA
P.S. Thiagarajan	National University of Singapore

Frits Vaandrager	Radboud University Nijmegen, The Netherlands
Frank Valencia	École Polytechnique de Paris, CNRS, France
Franck Van Breugel	York University, UK
Rob Van Glabbeek	NICTA, Australia
Nobuko Yoshida	Imperial College London, UK
Lijun Zhang	Technical University of Denmark

Additional Reviewers

Abbes, Samy
Abdulla, Parosh
Accattoli, Beniamino
Aceto, Luca
Ahmad, Arbob
Ancona, Davide
Atig, Mohamed Faouzi
Bacci, Giorgio
Bacci, Giovanni
Bakhshi, Rena
Barbanera, Franco
Bartoletti, Massimo
Beffara, Emmanuel
Bollig, Benedikt
Bonchi, Filippo
Bonelli, Eduardo
Bono, Viviana
Bourke, Timothy
Brenguier, Romain
Bruni, Roberto
Bujorianu, Marius
Buscemi, Marzia
Caltais, Georgiana
Chadha, Rohit
Chen, Xiwen
Chen, Yu-Fang
Cirstea, Corina
Corin, Ricardo
Courcelle, Bruno
Crafa, Silvia
Dardha, Ornela
David, Alexandre
de Frutos-Escrig, David
De Gouw, Stijn
De Nicola, Rocco
de Vink, Erik
Demangeon, Romain
Deng, Yuxin
Deniélou, Pierre-Malo
Derrick, John
Dhar, Amit Kumar
Dodds, Mike
Doyen, Laurent
Dubslaff, Clemens
Eisentraut, Christian
Elmas, Tayfun
Emmi, Michael
Engelbredt Dalsgaard, Andreas
Esparza, Javier
Fages, François
Fantechi, Alessandro
Feng, Yuan
Ferrer Fioriti, Luis María
Fleischhack, Hans
Fontana, Peter
Forejt, Vojtech
Froeschle, Sibylle
Gadducci, Fabio
Ganty, Pierre
Garbervetsky, Diego
Gebler, Daniel
Geeraerts, Gilles
Giachino, Elena
Giunti, Marco
Goltz, Ursula
Gutierrez, Julian
Habel, Annegret
Hahn, Ernst Moritz
Hashemi, Vahid
Hatefi, Hassan

Hayman, Jonathan
Haziza, Frédéric
Heckel, Reiko
Heljanko, Keijo
Helouet, Loic
Helvensteijn, Michiel
Hooman, Jozef
Howar, Falk
Jackson, Ethan K.
Jancar, Petr
Katoen, Joost-Pieter
Kiefer, Stefan
Klein, Joachim
Klueppelholz, Sascha
Knight, Sophia
Krcal, Jan
Kretinsky, Jan
Kucera, Antonin
Lanese, Ivan
Laneve, Cosimo
Laroussinie, Francois
Lazic, Ranko
Lee, Matias David
Leonardsson, Carl
Liang, Hongjin
Liu, Wanwei
Loeding, Christof
Loreti, Michele
Mardare, Radu
Markovski, Jasen
Marques, Eduardo R.B.
Martins, Francisco
Matteplackel, Raj Mohan
Minea, Marius
Mio, Matteo
Mousavi, Mohammadreza
Musuvathi, Madanlal
Møller, Mikael H.
Narayan, Kumar
Niebert, Peter
Noll, Thomas
Norman, Gethin
Novotný, Petr
Obdrzalek, Jan
Ong, Luke
Oshman, Rotem
Oskevskaya, Elena
Padovani, Luca
Palamidessi, Catuscia
Peters, Kirstin
Petri, Gustavo
Phillips, Iain
Pino, Luis
Pitcher, Corin
Place, Thomas
Pouzet, Marc
Prabhu, Vinayak
Pérez, Jorge A.
Randour, Mickael
Ranzato, Francesco
Rehak, Vojtech
Reniers, Michael
Reniers, Michel
Rensink, Arend
Rubin, Sasha
Ruppert, Eric
Saivasan, Prakash
Salaun, Gwen
Sankur, Ocan
Sassolas, Mathieu
Schicke-Uffmann, Jens-Wolfhard
Schlachter, Uli
Schmitt, Alan
Schmitz, Sylvain
Schwoon, Stefan
Silva, Alexandra
Song, Fu
Spieler, David
Srba, Jiri
Stahl, Christian
Strejcek, Jan
Sznajder, Nathalie
Tiezzi, Francesco
Tuosto, Emilio
Turrini, Andrea
Tzevelekos, Nikos
Ulidowski, Irek
Vafeiadis, Viktor
van Raamsdonk, Femke
Vandin, Andrea

Velner, Yaron
Vijzelaar, Stefan
Virbitskaite, Irina
Vogler, Walter
Wachter, Björn
Wang, Bow-Yaw
Wehrheim, Heike
Wimmel, Harro
Worrell, James
Xue, Bingtian
Yang, Shaofa
Zarko, Luke
Zavattaro, Gianluigi

Table of Contents

Session 1: Invited Talks

Session 2: Process Semantics and Modal Transition Systems

Session 3: VAS and Pushdown Systems

Session 7: Stochastic Models

Session 8: Message-Based Interacting Processes

Session 9: Principles of Automatic Verification

Session 10: Games and Control Synthesis

Reasoning with MAD Distributed Systems

Lorenzo Alvisi and Edmund L. Wong

Laboratory for Advanced Systems Research (LASR)
Department of Computer Science, The University of Texas at Austin
2317 Speedway, 2.302
Austin, TX 78712 USA
{lorenzo,elwong}@cs.utexas.edu

How does one reason about and build dependable distributed systems in which no component is guaranteed to follow the specified protocol?

While the setting of this question may appear implausible, this is precisely the environment in which services that span multiple administrative domains (MAD) must function. In such services—which include applications such as content dissemination (e.g.., [2]),, file backup (e.g., [6]), volunteer computing (e.g., [5]), multihop wireless networking (e.g., [4]), and Internet routing—resources are not under the control of a single administrative domain, so the necessary cooperation cannot simply be achieved by fiat. Instead, it is imperative that the service be structured so that nodes—which are administered by different, potentially selfish entities—have an incentive to help sustain it. Indeed, such issues are not imaginary: ample evidence suggests that a large number of peers will free-ride or deviate from the assigned protocol if it is in their interest to do so (e.g., [3,9,16,21]).

The presence of rational nodes challenges all approaches to dependability that rely on a clean separation between correct and faulty nodes. The standard approach to fault tolerance that relies on correct nodes to take appropriate action to mask or tolerate faulty nodes no longer applies when nodes that are not faulty may nonetheless selfishly deviate from their correct specification. Even the basic question of deciding on a failure model appropriate for reasoning about dependable MAD systems does not offer an obvious answer. Of course one could model all deviations, whether due to faults or to selfishness, as Byzantine faults [17], but many interesting problems in distributed computing become unsolvable once the number of Byzantine nodes exceeds a third of the total [17]—and there is no inherent reason why the combined number of faulty and selfish node should conveniently stay below that threshold. Alternatively, one could apply classic notions from game theory to model selfish behavior, but these typically only account for rational behavior, and become brittle if some (faulty) nodes behave in a seemingly irrational fashion. This is particularly the case in cooperative services, where the nodes themselves are often unreliable personal machines riddled with malware and other exploits [1,11]. The natural way forward, then, is to somehow combine insight from game theory and fault-tolerant distributed computing.

One strategy is to specify a notion of equilibrium that draws inspiration from traditional Byzantine fault tolerance and to aim for a solution concept in which rational nodes prefer the specified strategy despite the presence of a threshold of arbitrary failures [7,8,15]. This is the approach adopted by the elegant

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 1–4, 2013.

(k,t)-*robustness* solution concept [7]. Rational nodes have no incentive to deviate from a (k,t)-robust equilibrium despite up to t Byzantine nodes; further, unlike Nash equilibria, which are robust only against unilateral deviations, (k,t)-robust equilibria can tolerate collusions of up to k rational nodes. Unfortunately, the elegance of (k,t)-robustness comes at the cost of strong assumptions, which in principle can limit the practical applicability of this solution concept in realistic scenarios (more on this later).

An alternative strategy, explored in the BAR approach [10,14] is to classify nodes as belonging to one of three classes (Byzantine, Acquiescent, and Rational) and to model explicitly the expectations held by rational nodes concerning the behavior of Byzantine nodes. On the positive side, the BAR model has been successfully applied to build several real systems [10,18,19] that tolerate both malicious and rational deviations; however, these systems, as well as other work that has relied on models similar to BAR [20], suffer from several limitations of their own. First, they assume that rational nodes always model that Byzantine behavior as malicious, with Byzantine nodes hell-bent on producing the worst possible outcome for every other node; second, while they do not *rely* on acquiescent nodes to provide their guarantees, they also do not take advantage of their presence; finally, they do not explicitly handle collusion among rational nodes: at best, colluding rational nodes are modeled as Byzantine [10,19].

What should then be the basis for a rigorous treatment of cooperative services? How should participating nodes be modeled and what guarantees should we aim for? And can these models and guarantees be applied to real systems? This talk reviews our recent progress in trying to answer these questions.

Which solution concept can offer rigorous and practical basis for dependable cooperative services? To answer this question, we introduce a *communication game* that captures the key characteristics of most distributed systems that tolerate arbitrary faults. Specifically, our game models systems in which (a) some node-to-node communication is necessary to achieve some desired functionality (b) bandwidth is not free; and (c) the desired functionality is achievable despite t Byzantine failures. We find that notions of equilibrium inspired by traditional Byzantine fault-tolerant techniques, such as (k,t)-robustness, are capable of achieving equilibrium in communication games only under very limited circumstances, severely limiting their practical usefulness [22,24]. Our findings suggest that practical solution concepts must explicitly model the *beliefs* of rational nodes when it comes to Byzantine behavior.

What is the role of acquiescent nodes in cooperative services? Although real MAD systems include a sizable fraction of acquiescent (correct and unselfish) nodes, their impact on the incentive structure of MAD services is not well understood. In particular, systems built under the BAR model have sidestepped the challenge by designing protocols that neither depend on nor leverage the presence of acquiescent nodes—indeed, it seems possible that the very presence of acquiescent nodes may demotivate selfish rational nodes from contributing their share of resources, in the hope of free-riding off the acquiescent nodes' good will.

Can that good will be leveraged without imperiling rational participation? By distilling this question to a rational peer's *last* opportunity to cooperate, we find that not only is the good will of acquiescent nodes not antithetical to rational cooperation, but that, in a fundamental way, rational cooperation can only be achieved in the presence of the scintilla of altruism that acquiescent nodes bring to the system [23].

How should collusion be managed? The literature offers two approaches to guarantee that deviations resulting from collusion do not affect the incentives provided to rational nodes. The first is to model collusion as a fault and colluding nodes as Byzantine—which, similar to modeling rational deviations as Byzantine, forces an artificially low cap on the number of colluders. The second approach—taken by *strong Nash* [12], *k-resilient* equilibria [7,8], and *coalition-proof* Nash equilibria [13], to name a few—is to deny any benefit to colluders: if the equilibrium is a best response not just to every individual, but also to every possible coalition, then collusion poses no harm to the equilibrium's stability, since nodes gain no benefit by colluding. However, nodes that collude are likely to trust each other more and, more generally, be able to hold stronger assumptions about one another. Since stronger assumptions typically lead to more efficient protocols, identifying a single strategy that is a best response both inside and outside of every possible coalition is in practice very hard.

To overcome this challenge, we propose a fundamentally different approach to dealing with coalitions, based on the observation that while finding a single best response between all nodes is sufficient to prevent nodes from deviating, it is not *necessary* to achieve such stability. We introduce two new notions of equilibrium that leverage the observation that coalitions (including the trivial singleton coalition of one non-colluding node) will not deviate from an equilibrium as long as the equilibrium specifies a best-response strategy for every *coalition*. We thus allow the strategy a node follows to depend on whom the node is colluding with, thereby enabling the equilibrium to explicitly account for the advantages of coalition members while guaranteeing that nodes have no incentive to deviate from the specified equilibrium [22].

$\sim * \sim$

We are working on the design and implementation of a new hybrid (in that it relies on both servers and peer-to-peer cooperation) content distribution system that aims to apply these insights towards building a scalable, robust, and dependable method for distributing content.

References

1. 1 in 5 Macs has malware on it. Does yours?, `http://nakedsecurity.sophos.com/2012/04/24/mac-malware-study`
2. BitTorrent, `http://bittorrent.com`
3. Kazaa Lite, `http://en.wikipedia.org/wiki/Kazaa_Lite`
4. OpenGarden, `http://opengarden.com`
5. SETI@home, `http://setiathome.ssl.berkeley.edu`

6. SpaceMonkey, http://spacemonkey.com
7. Abraham, I., Dolev, D., Gonen, R., Halpern, J.: Distributed computing meets game theory: Robust mechanisms for rational secret sharing and multiparty computation. In: Proceedings of the 25th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (July 2006)
8. Abraham, I., Dolev, D., Halpern, J.Y.: Lower bounds on implementing robust and resilient mediators. In: Proceedings of the 5th IACR Theory of Cryptography Conference (March 2008)
9. Adar, E., Huberman, B.A.: Free riding on Gnutella. First Monday 5(10), 2–13 (2000), http://www.firstmonday.org/issues/issue5_10/adar/index.html
10. Aiyer, A.S., Alvisi, L., Clement, A., Dahlin, M., Martin, J.P., Porth, C.: BAR fault tolerance for cooperative services. In: Proceedings of the 20th ACM Symposium on Operating Systems Principles (October 2005)
11. Alyias, D., Batchelder, D., Blackbird, J., Faulhaber, J., Felstead, D., Henry, P., Jones, J., Kuo, J., Lauricella, M., Li, L., Ng, N., Rains, T., Sekhar, V., Stewart, H., Thomlinson, M., Zink, T.: Microsoft Security Intelligence Report, July through December, vol. 14 (2012)
12. Aumann, R.J.: Acceptable points in general cooperative n-person games. Annals of Mathematics Study 40(4), 287–324 (1959)
13. Bernheim, B.D., Peleg, B., Whinston, M.D.: Coalition-proof Nash equilibria, I. Concepts. Journal of Economic Theory 42(1), 1–12 (1987)
14. Clement, A., Li, H.C., Napper, J., Martin, J.P., Alvisi, L., Dahlin, M.: Bar primer. In: DSN, pp. 287–296 (2008)
15. Eliaz, K.: Fault tolerant implementation. Review of Economic Studies 69, 589–610 (2002)
16. Hughes, D., Coulson, G., Walkerdine, J.: Free riding on Gnutella revisited: The bell tolls? IEEE Distributed Systems Online 6(6) (June 2005)
17. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems 4(3), 382–401 (1982)
18. Li, H.C., Clement, A., Marchetti, M., Kapritsos, M., Robison, L., Alvisi, L., Dahlin, M.: FlightPath: Obedience vs. choice in cooperative services. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (December 2008)
19. Li, H.C., Clement, A., Wong, E.L., Napper, J., Roy, I., Alvisi, L., Dahlin, M.: BAR Gossip. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (November 2006)
20. Moscibroda, T., Schmid, S., Wattenhofer, R.: When selfish meets evil: Byzantine players in a virus inoculation game. In: Proceedings of the 25th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (July 2006)
21. Saroiu, S., Gummadi, P.K., Gribble, S.D.: A measurement study of peer-to-peer file sharing systems. In: Proceedings of the 9th Annual ACM/SPIE Multimedia Computing and Networking (January 2002)
22. Wong, E.L., Alvisi, L.: What's a little collusion between friends? In: Proceedings of the 32nd Annual ACM Symposium on Principles of Distributed Computing (July 2013)
23. Wong, E.L., Leners, J.B., Alvisi, L.: It's on me! The benefit of altruism in BAR environments. In: Proceedings of the 24th International Symposium on Distributed Computing (September 2010)
24. Wong, E.L., Levy, I., Alvisi, L., Clement, A., Dahlin, M.: Regret freedom isn't free. In: Proceedings of the 15th International Conference on Principles of Distributed Systems (December 2011)

The Power of Well-Structured Systems*

Sylvain Schmitz and Philippe Schnoebelen

LSV, ENS Cachan & CNRS, Cachan, France
{schmitz,phs}@lsv.ens-cachan.fr

Abstract. Well-structured systems, aka WSTS, are computational models where the set of possible configurations is equipped with a well-quasi-ordering which is compatible with the transition relation between configurations. This structure supports generic decidability results that are important in verification and several other fields.

This paper recalls the basic theory underlying well-structured systems and shows how two classic decision algorithms can be formulated as an exhaustive search for some "bad" sequences. This lets us describe new powerful techniques for the complexity analysis of WSTS algorithms. Recently, these techniques have been successful in precisely characterizing the power, in a complexity-theoretical sense, of several important WSTS models like unreliable channel systems, monotonic counter machines, or networks of timed systems.

Introduction

Well-Structured (Transition) Systems, aka WSTS, are a family of computational models where the usually infinite set of states is equipped with a well-quasi-ordering that is "compatible" with the computation steps. The existence of this well-quasi-ordering allows for the decidability of some important behavioural properties like Termination or Coverability.

Historically, the idea can be traced back to Finkel [21] who gave a first definition for WSTS abstracting from Petri nets and fifo nets, and who showed the decidability of Termination and Finiteness (aka Boundedness). Then Finkel [22] applied the WSTS idea to Termination of lossy channel systems, while Abdulla and Jonsson [4] introduced the backward-chaining algorithm for Coverability. One will find a good survey of these early results, and a score of WSTS examples, in [2, 24, 1, 8].

The basic theory saw several important developments in recent years, like the study of comparative expressiveness for WSTS [3], or the completion technique for forward-chaining in WSTS [23]. Simultaneously, many new WSTS models have been introduced (in distributed computing, software verification, or other fields), using well-quasi-orderings based on trees, sequences of vectors, or graphs (see references in [41]), rather than the more traditional vectors of natural numbers or words with the subword relation.

* Work supported by the ReacHard project, ANR grant 11-BS02-001-01.

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 5–24, 2013.

Another recent development is the complexity analysis of WSTS models and algorithms. New techniques, borrowing notions from proof theory and ordinal analysis, can now precisely characterize the complexity of some of the most widely used WSTS [13, 42, 27]. The difficulty here is that one needs complexity functions, complexity classes, and hard problems, with which computer scientists are not familiar.

The aim of this paper is to provide a gentle introduction to the main ideas behind the complexity analysis of WSTS algorithms. These complexity questions are gaining added relevance: more and more recent papers rely on reductions to (or from) a known WSTS problem to show the decidability (or the hardness) of problems in unrelated fields, from modal and temporal logic [31, 38] to XPath-like queries [29, 7].

Outline of the paper. Section 1 recalls the definition of WSTS, Sec. 2 illustrates it with a simple example, while Sec. 3 presents the two main verification algorithms for WSTS. Section 4 bounds the running time of these algorithms by studying the length of bad sequences using fast-growing functions. Section 5 explains how lower bounds matching these enormous upper bounds have been established in a few recent works, including the $\mathbf{F}_{\varepsilon_0}$-completeness result in this volume [26].

1 What Are WSTS?

A simple, informal way to define WSTS is to say that they are transition systems *whose behaviour is monotonic w.r.t. a well-ordering.* Here, monotonicity of behaviour means that the states of the transition system are ordered in a way such that larger states have more available steps than smaller states. Requiring that the ordering of states is a well-ordering (more generally, a well-quasi-ordering) ensures that monotonicity translates into decidability for some behavioural properties like Termination or Coverability.

Let us start with monotonicity. In its simplest form, a *transition system* (a TS) is a structure $\mathcal{S} = (S, \to)$ where S is the set of *states* (typical elements $s_1, s_2, \ldots$) and $\to \subseteq S \times S$ is the *transition relation.* As usual, we write "$s_1 \to s_2$" rather than "$(s_1, s_2) \in \to$" to denote steps. A TS is *ordered* when it is further equipped with a quasi-ordering of its states, i.e., a reflexive and transitive relation $\leq \subseteq S \times S$.

Definition 1.1 (Monotonicity). *An ordered transition system $\mathcal{S} = (S, \to, \leq)$ is* monotonic $\overset{\text{def}}{\Leftrightarrow}$ *for all $s_1, s_2, t_1 \in S$*

$$\big(s_1 \to s_2 \text{ and } s_1 \leq t_1\big) \text{ implies } \exists t_2 \in S : \big(t_1 \to t_2 \text{ and } s_2 \leq t_2\big)\,.$$

This property is also called "*compatibility*" (of the ordering with the transitions) [24]. Formally, is just means that $\leq$ is a *simulation* relation for $\mathcal{S}$, in precisely the classical sense of Milner [37]. The point of Def. 1.1 is to ensure that a "larger state" can do "more" than a smaller state. For example, it entails the following Fact that plays a crucial role in Sec. 3.

Given two finite runs $\boldsymbol{s} = (s_0 \to s_1 \to \cdots \to s_n)$ and $\boldsymbol{t} = (t_0 \to t_1 \to \cdots \to t_m)$, we say that $\boldsymbol{s}$ simulates $\boldsymbol{t}$ *from below* (and $\boldsymbol{t}$ simulates $\boldsymbol{s}$ *from above*) if $n = m$ and $s_i \leq t_i$ for all $i = 0, \ldots, n$.

Fact 1.2. *Any run $\boldsymbol{s} = (s_0 \to s_1 \to \cdots \to s_n)$ in a WSTS $\mathcal{S}$ can be simulated from above, starting from any $t \geq s_0$.*

Remark 1.3. Definition 1.1 comes in many variants. For example, Finkel and Schnoebelen [24] consider *strict compatibility* (when $<$, the strict ordering underlying $\leq$, is a simulation), *transitive compatibility* (when $\leq$ is a weak simulation), and the definition can further extend to *labeled* transition systems. These are all inessential variations of the main idea. □

Now to the wqo ingredient.

Definition 1.4 (Wqo). *A quasi-order $(S, \leq)$ is* well *("is a* wqo*") if every infinite sequence $s_0, s_1, s_2, \ldots$ over S contains an increasing pair $s_i \leq s_j$ for some $i < j$. Equivalently, $(S, \leq)$ is a wqo if, and only if, every infinite sequence $s_0, s_1, s_2, \ldots$ over S contains an infinite increasing subsequence $s_{i_0} \leq s_{i_1} \leq s_{i_2} \leq \cdots$, where $i_0 < i_1 < i_2 < \cdots$*

We call *good* a sequence that contains an increasing pair, otherwise it is *bad*. Thus in a wqo all infinite sequences are good, all bad sequences are finite.

Definition 1.4 offers two equivalent definitions. Many other characterisations exist [30], and it is an enlightening exercise to prove their equivalence [see 41, Chap. 1]. Let us illustrate the usefulness of the alternative definition: for a dimension $k \in \mathbb{N}$, write $\mathbb{N}^k$ for the set of k-tuples, or vectors, of natural numbers. For two vectors $\boldsymbol{a} = (a_1, \ldots, a_k)$ and $\boldsymbol{b} = (b_1, \ldots, b_k)$ in $\mathbb{N}^k$, we let $\boldsymbol{a} \leq_\times \boldsymbol{b} \stackrel{\text{def}}{\Leftrightarrow} a_1 \leq b_1 \wedge \cdots \wedge a_k \leq b_k$.

Example 1.5 (Dickson's Lemma). $(\mathbb{N}^k, \leq_\times)$ is a wqo.

Proof (of Dickson's Lemma). Consider an infinite sequence $\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3, \ldots$ over $\mathbb{N}^k$ and write $\boldsymbol{a}_i = (a_{i,1}, \ldots, a_{i,k})$. One can extract an infinite subsequence $\boldsymbol{a}_{i_1}, \boldsymbol{a}_{i_2}, \boldsymbol{a}_{i_3}, \ldots$ that is increasing over the first components, i.e., with $a_{i_1,1} \leq a_{i_2,1} \leq a_{i_3,1} \leq \cdots$, since $(\mathbb{N}, \leq)$ is a wqo (easy to prove, here the first definition suffices). From this infinite subsequence, one can further extract an infinite subsequence that is also increasing on the second components (again, using that $\mathbb{N}$ is wqo). After k extractions, one has an infinite subsequence that is increasing on all components, i.e., that is increasing for $\leq_\times$ as required. □

We can now give the central definition of this paper:

Definition 1.6 (WSTS). *An ordered transition system $\mathcal{S} = (S, \to, \leq)$ is a WSTS $\stackrel{\text{def}}{\Leftrightarrow}$ $\mathcal{S}$ is monotonic and $(S, \leq)$ is a wqo.*

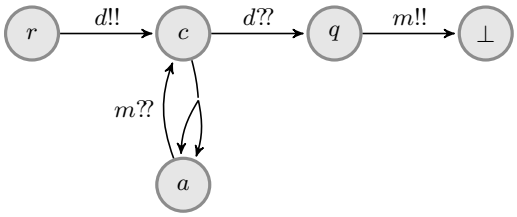


Fig. 1. A broadcast protocol

2 A Running Example: Broadcast Protocols

As a concrete illustration of the principles behind WSTS, let us consider distributed systems known as *broadcast protocols* [16, 17]. Such systems gather an unbounded number of identical finite-state processes running concurrently, able to spawn new processes, and communicating either via *rendez-vous*—where two processes exchange a message—or via *broadcast*—where one process sends the same message to every other process. While this may seem at first sight rather restricted for modeling distributed algorithms, broadcast protocols have been employed for instance to verify the correction of cache coherence protocols without fixing a number of participating processes.[1]

Formally, a broadcast protocol is defined as a triple $\mathtt{B} = (Q, M, R)$ where Q is a finite set of *locations*, M a finite set of *messages*, and R is a set of *rules*, that is, tuples (q, op, q') in $Q \times Op \times Q$, each describing an operation op available in the location q and leading to a new location q', where op can be a sending (denoted $m!$) or a receiving ($m?$) operation of a rendez-vous message m from M, or a sending ($m!!$) or receiving ($m??$) operation of a broadcast message m from M, or a spawning ($\mathrm{sp}(p)$) of a new process that will start executing from location p. As usual, we write $q \xrightarrow{op}_{\mathtt{B}} q'$ if (q, op, q') is in R.

Figure 1 displays a toy example where $Q = \{r, c, a, q, \bot\}$ and $M = \{d, m\}$: processes in location c can spawn new "active" processes in location a, while also moving to location a (a rule depicted as a double arrow in Fig. 1). These active processes are flushed upon receiving a broadcast of either m (emitted by a process in location q) or d (emitted by a process in location r); location $\bot$ is a sink location modeling process destruction.

The operational semantics of a broadcast protocol is expressed as a transition system $\mathcal{S}_{\mathtt{B}} = (S, \to)$, where states, here called configurations, are (finite) multisets of locations in Q, hence $S = \mathbb{N}^Q$. Informally, the intended semantics for a configuration s in $\mathbb{N}^Q$ is to record for each location q in Q the number of processes $s(q)$ currently in this location. We use a "sets with duplicates" notation, like $s = \{q_1, \ldots, q_n\}$ where some q_i's might be identical, and feel free to write, e.g., $\{q^3, q'^4\}$ instead of $\{q, q, q, q', q', q', q'\}$. A natural ordering for $\mathbb{N}^Q$ is the *inclusion* ordering defined by $s \subseteq s' \stackrel{\text{def}}{\Leftrightarrow} \forall q \in Q,\ s(q) \leq s'(q)$. For instance, $\{q, q, q'\} \subseteq \{q, q, q, q'\}$ but $\{q, q', q'\} \not\subseteq \{q, q, q'\}$ if $q \neq q'$. We further write

[1] See also the up-to-date survey of parameterized verification problems in [18].

$s = s_1 + s_2$ for the union (not the lub) of two multisets, in which case $s - s_1$ denotes s_2.

It remains to define how the operations of B update such a configuration through transitions $s \to s'$ of $\mathcal{S}_{\mathtt{B}}$:

rendez-vous step: if $q_1 \xrightarrow{m!}_{\mathtt{B}} q_1'$ and $q_2 \xrightarrow{m?}_{\mathtt{B}} q_2'$ for some $m \in M$, then $s + \{q_1, q_2\} \to s + \{q_1', q_2'\}$ for all s in $\mathbb{N}^Q$,

spawn step: if $q \xrightarrow{\mathrm{sp}(p)}_{\mathtt{B}} q'$, then $s + \{q\} \to s + \{q', p\}$ for all s in $\mathbb{N}^Q$,

broadcast step: if $q_0 \xrightarrow{m!!}_{\mathtt{B}} q_0'$ and $q_i \xrightarrow{m??}_{\mathtt{B}} q_i'$ for all $1 \leq i \leq k$ (and some m), then $s + \{q_0, q_1, \ldots, q_k\} \to s + \{q_0', q_1', \ldots, q_k'\}$ for all s in $\mathbb{N}^Q$ that do not contain a potential receiver for the broadcast, i.e., such that $s(q) = 0$ for all rules of the form $q \xrightarrow{m??}_{\mathtt{B}} q'$.

With the protocol of Fig. 1, the following steps are possible (with the spawned location or exchanged messages indicated on the arrows):

$$\{c^2, q, r\} \xrightarrow{a} \{a^2, c, q, r\} \xrightarrow{a} \{a^4, q, r\} \xrightarrow{m} \{c^4, r, \bot\} \xrightarrow{d} \{c, q^4, \bot\}.$$

We have just associated an ordered transition system $\mathcal{S}_{\mathtt{B}} = (\mathbb{N}^Q, \to, \subseteq)$ with every broadcast protocol B and are now ready to prove the following fact.

Fact 2.1. *Broadcast protocols are WSTS.*

Proof. First, $(\mathbb{N}^Q, \subseteq)$ is a wqo: since Q is finite, this is just another instance of Dickson's Lemma.

There remains to check that $\mathcal{S}_{\mathtt{B}}$ is monotonic. Formally, this is done by considering an arbitrary step $s_1 \to s_2$ (there are three cases) and an arbitrary pair $s_1 \subseteq t_1$. It is enough to assume that $t_1 = s_1 + \{q\}$, i.e., t_1 is just one location bigger that s_1, and to rely on transitivity. If $s_1 \to s_2$ is a rendez-vous step with $s_2 = s_1 - \{q_1, q_2\} + \{q_1', q_2'\}$, then $t_1 = s_1 + \{q\}$ also has a rendez-vous step $t_1 \to t_2 = t_1 - \{q_1, q_2\} + \{q_1', q_2'\}$ and one sees that $s_2 \subseteq t_2$ as required. If now if $s_1 \to s_2$ is a spawn step, a similar reasoning proves that $s_1 + \{q\} \to s_2 + \{q\}$. Finally, when $s_1 = s + \{q_1, \ldots\} \to s_2 = s + \{q_1', \ldots\}$ is a broadcast step, one proves that $s_1 + \{q\} \to s_2 + \{q'\}$ when there is a rule $q \xrightarrow{m??}_{\mathtt{B}} q'$, or when q is not a potential receiver and $q' = q$. □

Remark 2.2. One can show that the protocol depicted in Fig. 1 always terminates, this from any initial configuration s_{init}. Recall that, for a TS $\mathcal{S}$, Termination is the question, given a state $s_{init} \in S$, whether all runs starting from s_{init} are finite, i.e., whether there are no infinite runs from s_{init}.

A first remark is that the processes in location $\bot$ can safely be ignored, since they have terminated. Then, consider any sequence of steps $s_0 \to s_1 \to \cdots \to s_i \to \cdots$ with each configuration of form $s_i = \{a^{n_{a,i}}, c^{n_{c,i}}, q^{n_{q,i}}, r^{n_{r,i}}\}$, and let us compare two configurations indexed by $i < j$:

– either only spawn steps occur between s_i and s_j, thus $n_{c,j} < n_{c,i}$,

- or at least one m has been broadcast but no d has been broadcast, thus $n_{q,j} < n_{q,i}$,
- or at least one d has been broadcast, and then $n_{r,j} < n_{r,i}$.

Thus in all cases, $s_i \not\subseteq s_j$, i.e. the sequence of successive configurations is bad. Since $(\mathbb{N}^Q, \subseteq)$ is a wqo, there is no infinite bad sequence, hence no infinite run. □

3 Verification of WSTS

In this section we present the two main generic decision algorithms for WSTS. We strive for a presentation that abstracts away from implementation details, and that can directly be linked to the complexity analysis we describe in the following sections. The general idea is that these algorithms can be seen as an exhaustive search for some kind of bad sequences.

3.1 Termination

There is a generic algorithm deciding Termination on WSTS. The algorithm has been adapted and extended to show decidability of Inevitability (of which Termination is a special case), Finiteness (aka Boundedness), or Regular Simulation, see [2, 24].

Lemma 3.1 (Finite Witnesses for Infinite Runs). *A WSTS $\mathcal{S}$ has an infinite run from s_{init} if, and only if, it has a finite run from s_{init} that is a good sequence.*

Proof. Obviously, any infinite run $s_{init} = s_0 \to s_1 \to s_2 \to \cdots$ is a good sequence by property of $\leq$ being a wqo (see Def. 1.4). Once a pair $s_i \leq s_j$ is identified, the finite prefix that stops at s_j is both a finite run and a good sequence.

Reciprocally, given a finite run $s_0 \to s_1 \to \cdots \to s_i \to \cdots \to s_j$ with $i < j$ and $s_i \leq s_j$, Fact 1.2 entails the existence of a run $s_j \to s_{j+1} \to \cdots \to s_{2j-i}$ that simulates $s_i \to \cdots \to s_j$ from above. Hence the finite run can be extended to some $s_0 \to \cdots \to s_i \to \cdots \to s_j \to \cdots \to s_{2j-i}$ with $s_i \leq s_{2j-i}$. Repeating this extending process ad infinitum, one obtains an infinite run. □

Very little is needed to turn Lemma 3.1 into a decidability proof for Termination. We shall make some minimal *effectiveness assumptions*: (EA1) the set of states S is recursive; (EA2) the function $s \mapsto \mathit{Post}(s)$, that associates with any state its image by the relation $\to$, is computable (and image-finite, aka finitely branching); and (EA3) the wqo $\leq$ is decidable. We say that $\mathcal{S}$ is an *effective* WSTS when all three assumptions are fulfilled. Note that (EA1–2) hold of most computational models, starting with Turing machines and broadcast protocols, but we have to spell out these assumptions at some point since Def. 1.6 is abstract and does not provide any algorithmic foothold.

We can now prove the decidability of Termination for effective WSTS. Assume $\mathcal{S} = (S, \to, \leq)$ is effective. We are given some starting state $s_{init} \in S$. The existence of an infinite run is semi-decidable since infinite runs admit finite witnesses

by Lemma 3.1. (Note that we rely on all three effectiveness assumptions to guess a finite sequence $(s_{init} =)s_0, \ldots, s_i, \ldots, s_j$ of states, check that it is indeed a run of $\mathcal{S}$, and that it is indeed a good sequence.) Conversely, if all runs from s_{init} are finite, then there are only finitely many of them (by Kőnig's Lemma, since $\mathcal{S}$ is finitely branching), and it is possible to enumerate all these runs by exhaustive simulation, thanks to (EA2). Thus Termination is semi-decidable as well. Finally, since the Termination problem and its complement are both semi-decidable, they are decidable.

3.2 Coverability

After Termination, we turn to the decidability of Coverability, a slightly more involved result that is also more useful for practical purposes: the decidability of Coverability opens the way to the verification of safety properties and many other properties defined by fixpoints, see [8].

Recall that, for an ordered TS $\mathcal{S}$, Coverability is the question, given a starting state $s_{init} \in S$ and a target state $t \in S$, whether there is a run from s_{init} that eventually covers t, i.e., whether there is some s reachable from s_{init} with $s \geq t$. We call any finite run $s_0 \to s_1 \to \cdots \to s_n$ s.t. $s_n \geq t$ a *covering run (for t)*.

Rather than using covering runs to witness Coverability, we shall use "pseudoruns". Formally, a *pseudorun* is a sequence $s_0, \ldots, s_n$ such that, for all $i = 1, \ldots, n$, s_{i-1} can cover s_i *in one step*, i.e., $s_{i-1} \to t_i$ for some $t_i \geq s_i$. In particular, any run is also a pseudorun. And the existence of a pseudorun $s_0, \ldots, s_n$ with $s_n \geq t$ witnesses the existence of covering runs from any $s_{init} \geq s_0$ (proof: by repeated use of Fact 1.2).

A pseudorun $s_0, \ldots, s_n$ is *minimal* if, for all $i = 1, \ldots, n$, s_{i-1} is minimal among all the states from where s_i can be covered in one step (we say that s_{i-1} is a *minimal pseudopredecessor* of s_i).

Lemma 3.2 (Minimal Witnesses for Coverability). *If $\mathcal{S}$ has a covering run from s_{init}, it has in particular a minimal pseudorun $s_0, s_1, \ldots, s_n$ with $s_0 \leq s_{init}$, $s_n = t$, and such that the* reverse sequence *$s_n, s_{n-1}, \ldots, s_0$ is bad (we say that $s_0, \ldots, s_n$ is "revbad").*

Proof. Assume that $s_{init} = s_0 \to s_1 \to \cdots \to s_n$ is a covering run. Replacing s_n by t gives a pseudorun ending in t. We now show that if the pseudorun is not minimal or not revbad, then there is a "smaller" pseudorun.

First, assume that $s_n, s_{n-1}, \ldots, s_0$ is not bad. Then $s_i \geq s_j$ for some $0 \leq i < j \leq n$ and $s_0, s_1, \ldots, s_{i-1}, s_j, s_{j+1}, \ldots, s_n$ is again a pseudorun, shorter in length (note that $s_n = t$ is unchanged, while s_0 may have been replaced by a smaller s_j in the case where $i = 0$). If now $s_0, s_1, \ldots, s_n$ is not minimal, i.e., if some s_{i-1} is not a minimal pseudopredecessor of s_i, we may replace s_{i-1} by some other pseudopredecessor $s'_{i-1} \leq s_{i-1}$ that is minimal (since $(S, \leq)$ is wqo, hence well-founded, its non-empty subsets do have minimal elements) and $s_0, \ldots, s_{i-2}, s'_{i-1}, s_i, \ldots, s_n$ is again a pseudorun (where $s_n = t$ as before and where s_0 may have been replaced by a smaller s'_0). Repeating such shortening and lowering replacements as long as possible is bound to terminate (after at most

polynomially many replacements). The pseudorun we end up with is minimal, revbad, has $s_0 \leq s_{init}$ and $s_n = t$ as claimed. □

Turning Lemma 3.2 into a decidability proof is similar to what we did for Termination. This time we make the following effectiveness assumptions: (EA1) and (EA3) as above, with (EA2') the assumption that the function *MinPPre*—that associates with any state its finite set of minimal pseudopredecessors—is computable.[2] The set of all minimal revbad pseudoruns ending in t is finite (Kőnig's Lemma again: finite branching of the tree is ensured by minimality of the pseudoruns, while finite length of the branches is ensured by the restriction to revbad pseudoruns). This set of pseudoruns can be built effectively, starting from t and applying *MinPPre* repeatedly, but it is enough to collect the states that occur along them, using a standard backward-chaining scheme. We write $\mathit{MinPPre}^*(t)$ to denote the set of all these states: once they have been computed, it only remains to be checked whether s_{init} is larger than one of them, using (EA3) once more.

We conclude by observing that assumption (EA2'), though less natural-looking than (EA2), is satisfied in most computational models. As Ex. 4.1 shows for the case of broadcast protocols, computing $\mathit{MinPPre}(s)$ is often a simple case of finding the minimal solutions to a simple inverse problem on rewrite rules.

3.3 What Is the Complexity of WSTS Verification?

One aspect of the algorithms given in this section we have swept under the rug is how expensive they can be. This will be the topic of the next two sections, but as an appetizer, let us consider how long the Termination algorithm can run on the broadcast protocol of Fig. 1.

Let us ignore the number of processes in the sink location $\bot$. The protocol from Fig. 1 allows the following steps when spawn steps are performed as long as possible before broadcasting m:

$$\{c^n, q\} \xrightarrow{a^n} \{a^{2n}, q\} \xrightarrow{m} \{c^{2n}\} .$$

Such a greedy sequence of message thus doubles the number of processes in c and removes one single process from q. Iterating such sequences as long as some process is in q before broadcasting d then leads to:

$$\{c^{2^0}, q^n, r\} \xrightarrow{a^{2^0} m} \{c^{2^1}, q^{n-1}, r\} \xrightarrow{a^{2^1} m} \{c^{2^2}, q^{n-2}, r\}$$
$$\cdots \to \{c^{2^{n-1}}, q, r\} \xrightarrow{a^{2^{n-1}} m} \{c^{2^n}, r\} \xrightarrow{d} \{c^{2^0}, q^{2^n}\} .$$

[2] Recall that any subset of a wqo has only finitely many minimal elements up to the equivalence given by $s \equiv s' \overset{\text{def}}{\Leftrightarrow} s \leq s' \leq s$.

Such iterations thus implement an exponentiation of the number of processes in q in exchange for decrementing the number of processes in r by one. Repeating this kind of sequences therefore allows:

$$\{c, q, r^n\} \to^* \{c, q^{\mathrm{tower}(n)}\},$$

where $\mathrm{tower}(0) \stackrel{\mathrm{def}}{=} 1$ and $\mathrm{tower}(n+1) \stackrel{\mathrm{def}}{=} 2^{\mathrm{tower}(n)}$: although it always terminates, the broadcast protocol of Fig. 1 can exhibit sequences of steps of non-elementary length. This also entails a non-elementary lower bound on the Termination algorithm when run on this protocol: since the system terminates, all the runs need to be checked, including this particular non-elementary one. □

4 Upper Bounds on Complexity

Although the theory of well-structured systems provides generic algorithms for numerous verification problems, it might seem rather unclear, what the computational cost of running these algorithms could be—though we know their complexity can be considerable (recall Sec. 3.3). Inspecting the termination arguments in Sec. 3, we see that the critical point is the finiteness of bad sequences. Unfortunately, the wqo definition does not mention anything about the *length* of such sequences, but merely asserts that they are finite.

It turns out that very broadly applicable hypotheses suffice in order to define a maximal length for bad sequences (Sec. 4.1), which then gives rise to so-called *length function theorems* bounding such lengths using ordinal-indexed functions (Sec. 4.2). These upper bounds allow for a classification of the power of many WSTS models in complexity-theoretic terms (Sec. 4.3), and also lead to simplified WSTS algorithms that take advantage of the existence of computable upper bounds on the length of bad sequences (Sec. 4.4).

4.1 Controlled Sequences

The Length of Bad Sequences. If we look at a very simple quasi-order, namely $(Q, =)$ with a finite support Q and equality as ordering—which is a wqo by the pigeonhole principle—, we can only exhibit bad sequences with length up to $\#_Q$, the cardinality of Q. But things start going awry as soon as we consider infinite wqos; for instance

$$n, n-1, n-2, \ldots, 0 \tag{S1}$$

is a bad sequence over $(\mathbb{N}, \leq)$ for every n in $\mathbb{N}$, i.e. the length of a bad sequence over $(\mathbb{N}, \leq)$ can be arbitrary. Even if we restrict ourselves to bad sequences where the first element is not too large, we can still build arbitrarily long sequences: for instance, over $(\mathbb{N}^Q, \subseteq)$ with $Q = \{p, q\}$,

$$\{p\}, \{q^n\}, \{q^{n-1}\}, \ldots, \{q\}, \emptyset \tag{S2}$$

is a bad sequence of length $n + 2$.

Controlling Sequences. Here is however a glimpse of hope: the sequence (S2) cannot be the run of a broadcast protocol, due to the sudden "jump" from $\{p\}$ to an arbitrarily large configuration $\{q^n\}$. More generally, the key insight is that, in an algorithm that relies on a wqo for termination, successive states cannot jump to arbitrarily large sizes, because these states are constructed algorithmically: we call such sequences *controlled*.

Let $(A, \leq_A)$ be a wqo. Formally, we posit a *norm* $|.|_A : A \to \mathbb{N}$ on the elements of our wqo, which we require to be *proper*, in that only finitely many elements have norm n: put differently, $A_{\leq n} \stackrel{\text{def}}{=} \{x \in A \mid |x|_A \leq n\}$ must be finite for every n. For instance, $|s|_{\mathbb{N}^Q} \stackrel{\text{def}}{=} \max_{q \in Q} s(q)$ is a proper norm for $\mathbb{N}^Q$.

Given an increasing *control function* $g : \mathbb{N} \to \mathbb{N}$, we say that a sequence $x_0, x_1, \ldots$ over A is (g, n_0)-*controlled* if the norm of x_i is no larger that the ith iterate of g applied to n_0: $|x_i|_A \leq g^i(n_0)$ for all i. Thus g bounds the growth of the elements in the sequence, and n_0 is a bound on the initial norm $|x_0|_A$.

Example 4.1 (Controlled Successors in a Broadcast Protocol). Let us see how these definitions work on broadcast protocols. First, on the sequences of successors built in the Termination algorithm: If $s \to s'$ is a rendez-vous step, then $|s'|_{\mathbb{N}^Q} \leq 2 + |s|_{\mathbb{N}^Q}$, corresponding to the case where the two processes involved in the rendez-vous move to the same location. If $s \to s'$ is a broadcast step, then $|s'|_{\mathbb{N}^Q} \leq \#_Q \cdot |s|_{\mathbb{N}^Q}$, corresponding to the case where all the processes in s (of which there are at most $\#_Q \cdot |s|_{\mathbb{N}^Q}$) enter the same location. Finally, spawn steps only incur $|s'|_{\mathbb{N}^Q} \leq |s|_{\mathbb{N}^Q} + 2$. Thus $g(n) \stackrel{\text{def}}{=} \#_Q \cdot n$ defines a control function for any run in a broadcast protocol with $\#_Q \geq 2$ locations, provided the initial norm n_0 is chosen large enough. □

Example 4.2 (Controlled Minimal Pseudopredecessors in a Broadcast Protocol). Now for the minimal pseudopredecessors built in the course of the Coverability algorithm: Assume $|t|_{\mathbb{N}^Q} \leq n$ and $s \to s' \geq t$ is a step from some minimal s:

rendez-vous step: If $s' = (s - \{q_1, q_2\} + \{q'_1, q'_2\})$ in a rendez-vous, then
- either $\{q'_1, q'_2\} \subseteq t$ and thus $s = t - \{q'_1, q'_2\} + \{q_1, q_2\}$ and $|s|_{\mathbb{N}^Q} \leq n$,
- or $q'_i \notin t$ for exactly one i among $\{1, 2\}$, hence $s = t - \{q_{1-i}\} + \{q_1, q_2\}$ and $|s|_{\mathbb{N}^Q} \leq n + 1$,
- or $q'_i \notin t$ for any $i \in \{1, 2\}$ and $|s|_{\mathbb{N}^Q} \leq n + 2$ (note however that $s \subseteq t$ in this case and the constructed sequence would not be bad).

broadcast step: Assume $s' = (s - \{q_0, q_1, \ldots, q_k\} + \{q'_0, q'_1, \ldots, q'_k\})$ with $q_0 \xrightarrow{m!!}_{\mathtt{B}} q'_0$ the corresponding broadcast send rule. Because s is minimal, $\{q'_1, \ldots, q'_k\} \subseteq t$, as otherwise a smaller s could be used. Hence,
- either $q'_0 \in t$, and then $s = t - \{q'_0, q'_1, \ldots, q'_k\} + \{q_0, q_1, \ldots, q_k\}$ and $|s|_{\mathbb{N}^Q} \leq n$,
- or $q'_0 \notin t$, and then $s = t - \{q'_1, \ldots, q'_k\} + \{q_0, q_1, \ldots, q_k\}$ and $|s|_{\mathbb{N}^Q} \leq n + 1$.

spawn step: similar to a rendez-vous step, $|s|_{\mathbb{N}^Q} \leq n + 1$.

Therefore, $g(n) \stackrel{\text{def}}{=} n + 2$ defines a control function for any sequence of minimal pseudopredecessor steps in any broadcast protocol. □

Length Functions. The upshot of these definitions is that, unlike in the uncontrolled case, there is a longest (g, n_0)-controlled bad sequence over any normed wqo $(A, \leq_A)$: indeed, we can organize such sequences in a tree by sharing common prefixes; this tree has

- finite branching, bounded by the cardinal of $A_{\leq g^i(n_0)}$ for a node at depth i, and
- no infinite branches thanks to the wqo property.

By Kőnig's Lemma, this tree of bad sequences is therefore finite, of some height $L_{g,n_0,A}$ representing the length of the maximal (g, n_0)-controlled bad sequence(s) over A. In the following, since we are mostly interested in this length as a function of the initial norm n_0, we will see this as a *length function* $L_{g,A}(n)$; our purpose will then be to obtain complexity bounds on $L_{g,A}$ depending on g and A.

4.2 Length Function Theorems

Now that we are empowered with a suitable definition for the maximal length $L_{g,A}(n)$ of (g, n)-controlled bad sequences over A, we can try our hand at proving *length function theorems*, which provide constructible functions bounding $L_{g,A}$ for various normed wqos $(A, \leq)$. Examples of length function theorems can be found

- in [36, 15, 20, 5, 41] for Dickson's Lemma, i.e. for $(\mathbb{N}^Q, \subseteq)$ for some finite Q, which is isomorphic to $(\mathbb{N}^{\#_Q}, \leq_\times)$,
- in [5] for $(\mathcal{P}_f(\mathbb{N}^d), \preceq)$ the set of finite subsets of $\mathbb{N}^d$ with the majoring ordering defined by $X \preceq Y \overset{\text{def}}{\Leftrightarrow} \forall x \in X, \exists y \in Y, x \leq_\times y$,
- in [45, 14, 40] for Higman's Lemma, i.e. for $(\Sigma^*, \leq_*)$ the set of finite sequences over a finite alphabet Σ with the subword embedding $\leq_*$,
- in [45] for Kruskal's Tree Theorem, i.e. for $(T, \leq_T)$ the set of finite unranked ordered trees with the homeomorphic embedding $\leq_T$.

These theorems often differ in the hypotheses they put on g, the tightness of the upper bounds they provide, and on the simplicity of their proofs (otherwise the results of Weiermann [45] for Kruskal's Tree Theorem would include all the others). We will try to convey the flavour of the theorems from [40, 41] here.

Starting again with the case of a finite wqo $(Q, =)$, and setting $|x|_Q \overset{\text{def}}{=} 0$ for all x in Q as the associated norm, we find immediately that, for all g and n,

$$L_{g,Q}(n) = \#_Q \tag{1}$$

by the pigeonhole principle. Another easy example is $(\mathbb{N}, \leq)$ with norm $|k|_\mathbb{N} \overset{\text{def}}{=} k$:

$$L_{g,\mathbb{N}}(n) = n + 1 \,, \tag{2}$$

the bad sequence (S1) being maximal.

We know however from Sec. 3.3 that very long bad sequences can be constructed, so we should not hope to find such simple statements for $(\mathbb{N}^Q, \subseteq)$ and

more complex wqos. The truth is that the "tower" function we used in Sec. 3.3 is really benign compared to the kind of upper bounds provided by length function theorems. Such functions of enormous growth have mainly been studied in the context of subrecursive hierarchies and reverse mathematics (see e.g. [43, Chap. 4] for further reference); let us summarily present them.

Ordinal Indexed Functions. An idea in order to build functions of type $\mathbb{N} \to \mathbb{N}$ with faster and faster growths is to iterate smaller functions a number of times that depends on the argument—this is therefore a form of diagonalisation. In order to keep track of the diagonalisations, we can index the constructed functions with ordinals, so that diagonalisations occur at limit ordinals.

Ordinal Terms. First recall that ordinals α below ε_0 can be denoted as terms in *Cantor Normal Form*, aka CNF:

$$\alpha = \omega^{\beta_1} \cdot c_1 + \cdots + \omega^{\beta_n} \cdot c_n \text{ where } \alpha > \beta_1 > \cdots > \beta_n \text{ and } \omega > c_1, \ldots, c_n > 0 \,.$$

In this representation, $\alpha = 0$ if and only if $n = 0$. An ordinal with CNF of the form $\alpha' + 1$ (i.e. with $n > 0$ and $\beta_n = 0$) is called a *successor* ordinal, and otherwise if $\alpha > 0$ it is called a *limit* ordinal, and can be written as $\gamma + \omega^\beta$ by setting $\gamma = \omega^{\beta_1} \cdot c_1 + \cdots + \omega^{\beta_n} \cdot (c_n - 1)$ and $\beta = \beta_n$. We usually write "λ" to denote a limit ordinal.

A *fundamental sequence* for a limit ordinal λ is a sequence $(\lambda(x))_{x<\omega}$ of ordinals with supremum λ, with a standard assignment defined inductively by

$$(\gamma + \omega^{\beta+1})(x) \stackrel{\text{def}}{=} \gamma + \omega^\beta \cdot (x+1) \,, \qquad (\gamma + \omega^\lambda)(x) \stackrel{\text{def}}{=} \gamma + \omega^{\lambda(x)} \,. \tag{3}$$

This is one particular choice of a fundamental sequence, which verifies e.g. $0 < \lambda(x) < \lambda(y)$ for all $x < y$. For instance, $\omega(x) = x + 1$, $(\omega^{\omega^4} + \omega^{\omega^3+\omega^2})(x) = \omega^{\omega^4} + \omega^{\omega^3+\omega\cdot(x+1)}$.

Hardy Hierarchy. Let $h\colon \mathbb{N} \to \mathbb{N}$ be an increasing function. The *Hardy hierarchy* $(h^\alpha)_{\alpha<\varepsilon_0}$ controlled by h is defined inductively by

$$h^0(x) \stackrel{\text{def}}{=} x \,, \qquad h^{\alpha+1}(x) \stackrel{\text{def}}{=} h^\alpha(h(x)) \,, \qquad h^\lambda(x) \stackrel{\text{def}}{=} h^{\lambda(x)}(x) \,. \tag{4}$$

Observe that h^k for some finite k is the kth iterate of h (by using the first two equations solely). This intuition carries over: h^α is a transfinite iteration of the function h, using diagonalisation to handle limit ordinals. For instance, starting with the successor function $H(x) \stackrel{\text{def}}{=} x + 1$, we see that a first diagonalisation yields $H^\omega(x) = H^{x+1}(x) = 2x + 1$. The next diagonalisation occurs at $H^{\omega\cdot 2}(x) = H^{\omega+x+1}(x) = H^\omega(2x+1) = 4x + 3$. Fast-forwarding a bit, we get for instance a function of exponential growth $H^{\omega^2}(x) = 2^{x+1}(x+1) - 1$, and later a non elementary function H^{ω^3}, an "Ackermannian" non primitive-recursive function H^{ω^ω}, and an "hyper-Ackermannian" non multiply-recursive function $H^{\omega^{\omega^\omega}}$. Hardy functions are well-suited for expressing large iterates of a control function, and therefore for bounding the norms of elements in a controlled bad sequence.

Cichoń Hierarchy. A variant of the Hardy functions is the *Cichoń hierarchy* $(h_\alpha)_{\alpha<\varepsilon_0}$ controlled by h [14], defined by

$$h_0(x) \stackrel{\text{def}}{=} 0\,, \qquad h_{\alpha+1}(x) \stackrel{\text{def}}{=} 1 + h_\alpha\,(h(x))\,, \qquad h_\lambda(x) \stackrel{\text{def}}{=} h_{\lambda(x)}(x)\,. \tag{5}$$

For instance, $h_d(x) = d$ for all finite d, thus $h_\omega(x) = x+1$ regardless of the choice of the function h. One can check that $H^\alpha(x) = H_\alpha(x) + x$ when employing the successor function H; in general $h^\alpha(x) \geq h_\alpha(x) + x$ since h is assumed to be increasing.

This is the hierarchy we are going to use for our statements of length function theorems: a Hardy function h^α is used to bound the maximal norm of an element in a bad sequence, and the corresponding Cichoń function h_α bounds the length of the bad sequence itself, the two functions being related for all h, α, and x by

$$h^\alpha(x) = h^{h_\alpha(x)}(x)\,. \tag{6}$$

Length Functions for Dickson's Lemma. We can now provide an example of a length function theorem for a non-trivial wqo: Consider $(\mathbb{N}^Q, \subseteq)$ and some control function g. Here is one of the parametric bounds proved in [41, Chap. 2]:[3]

Theorem 4.3 (Parametric Bounds for Dickson's Lemma). *If $x_0, \ldots, x_L$ is a (g,n)-controlled bad sequence over $(\mathbb{N}^Q, \subseteq)$ for some finite set Q, then $L \leq L_{g,\mathbb{N}^Q}(n) \leq h_{\omega^{\#_Q}}(n)$ for the function $h(x) \stackrel{\text{def}}{=} \#_Q \cdot g(x)$.*

By Equation (6), we also deduce that the norm of the elements x_i in this bad sequence cannot be larger than $h^{\omega^{\#_Q}}(n)$.

A key property of such bounds expressed with Cichoń and Hardy functions is that they are *constructible* with negligible computational overhead (just apply their definition on a suitable encoding of the ordinals), which means that we can employ them in algorithms (see Sec. 4.4 for applications).

Given how enormous the Cichoń and Hardy functions can grow, it is reasonable at this point to ask how tight the bounds provided by Thm. 4.3 really are. At least in the case $\#_Q = 1$, we see these bounds match (2) since $h_\omega(n) = n+1$. We will show in Sec. 5 that similarly enormous complexity lower bounds can be proven for Termination or Coverability problems on WSTS, leaving only inessential gaps with the upper bounds like Thm. 4.3. In fact, we find such parametric bounds to be overly precise, because we would like to express simple *complexity* statements about decision problems.

4.3 Fast Growing Complexy Classes

As witnessed in Sec. 3.3 and the enormous upper bounds provided by Thm. 4.3, we need to deal with non-elementary complexities. The corresponding non-elementary complexity classes are arguably missing from most textbooks and

[3] A more general version of Thm. 4.3 in [40] provides $h_{o(A)}(n)$ upper bounds for (g,n)-controlled bad sequences over wqos $(A, \leq)$ constructed through disjoint unions, cartesian products, and Kleene star operations, where $o(A)$ is the maximal order type of $(A, \leq)$, i.e. the order type of its maximal linearization, and h is a low-degree polynomial in g. This matches Thm. 4.3 because $o(\mathbb{N}^Q) = \omega^{\#_Q}$.

references on complexity. For instance, the *Complexity Zoo*[4], an otherwise very richly populated place, features no intermediate steps between Elementary and the next class, namely Primitive-Recursive (aka PR), and a similar gap occurs between PR and Recursive (aka R). If we are to investigate the complexity of decision problems on WSTSs, much more fine-grained hierarchies are required.

Here, we present a hierarchy of ordinal-indexed *fast growing* complexity classes $(\mathbf{F}_\alpha)_\alpha$ tailored to *completeness* proofs for non-elementary problems [see 41, App. B]. When exploring larger complexities, the hierarchy includes non primitive-recursive classes, for which quite a few complete problems have arisen in the recent years, e.g. $\mathbf{F}_\omega$ in [35, 28, 44, 42, 19, 10, 33], $\mathbf{F}_{\omega^\omega}$ in [13, 38, 32, 6, 12, 7], $\mathbf{F}_{\omega^{\omega^\omega}}$ in [27], and $\mathbf{F}_{\varepsilon_0}$ in [26].

Let us define the classes of reductions and of problems we will consider:

$$\mathscr{F}_\alpha \stackrel{\text{def}}{=} \bigcup_{c<\omega} \text{FDTime}\left(H^{\omega^\alpha \cdot c}(n)\right), \quad \mathbf{F}_\alpha \stackrel{\text{def}}{=} \bigcup_{p \in \bigcup_{\beta<\alpha} \mathscr{F}_\beta} \text{DTime}\left(H^{\omega^\alpha}(p(n))\right). \tag{7}$$

The hierarchy of function classes $(\mathscr{F}_\alpha)_{\alpha \geq 2}$ is the *extended Grzegorczyk hierarchy* [34], and provides us with classes of non-elementary reductions: for instance $\mathscr{F}_2$ is the set of elementary functions, $\bigcup_{\beta<\omega} \mathscr{F}_\beta$ that of primitive-recursive functions, and $\bigcup_{\beta<\omega^\omega} \mathscr{F}_\beta$ that of multiply-recursive functions. The hierarchy of complexity classes $(\mathbf{F}_\alpha)_{\alpha \geq 3}$ features for instance a class $\mathbf{F}_\omega$ of Ackermannian problems closed under primitive-recursive reductions, and a class $\mathbf{F}_{\omega^\omega}$ of hyper-Ackermannian problems closed under multiply-recursive reductions. Intuitively, $\mathbf{F}_\omega$-complete problems are not primitive-recursive, but only *barely* so, and similarly for the other levels.

4.4 Combinatory Algorithms

Theorem 4.3 together with Ex. 4.1 (resp. 4.2) provides complexity upper bounds on the Termination (resp. Coverability) algorithm when applied to broadcast protocols. Indeed, a nondeterministic program can guess a witness of (non-) Termination (resp. Coverability), which is of length bounded by $L_{g,\mathbb{N}^Q}(n) + 1$, where g was computed in Ex. 4.1 (resp. 4.2) and n is the size of the initial configuration s_{init} (resp. target configuration t). By Thm. 4.3 such a witness has length bounded by $h_{\omega^{\#_Q}}(n)$ for $h(n) = \#_Q \cdot g(n)$, and by (6), the norm of the elements along this sequence is bounded by $h^{\omega^{\#_Q}}(n)$. Thus this non-deterministic program only needs space bounded by $\#_Q \cdot \log(h^{\omega^{\#_Q}}(n)) \leq H^{\omega^\omega}(p(n + \#_Q))$ for some primitive-recursive function p. Hence:

Fact 4.4. *Termination and Coverability of broadcast protocols are in* $\mathbf{F}_\omega$.

Thanks to the upper bounds on the length of bad sequences, the algorithms sketched above are really *combinatory* algorithms: they compute a maximal length for a witness and then nondeterministically check for its existence. In

[4] `https://complexityzoo.uwaterloo.ca`

the case of Termination, we can even further simplify the algorithm: if a run starting from s_{init} has length $> L_{g,A}(n)$, this run is necessarily a good sequence and the WSTS does not terminate.

5 Lower Bounds on Complexity

When considering the mind-numbling complexity upper bounds that come with applications of the Length Function Theorems from Sec. 4 to the algorithms of Sec. 3, a natural question that arises is whether this is the complexity one gets when using the rather simplistic Coverability algorithm from Sec. 3.2, or whether it is the intrinsic complexity of the Coverability *problem* for a given WSTS model.

There is no single answer here: for instance, on Petri nets, a breadth-first backward search for a coverability witness actually works in 2ExpTime [9] thanks to bounds on the size of minimal witnesses due to Rackoff [39]; on the other hand, a depth-first search for a termination witness can require Ackermannian time [11], this although both problems are ExpSpace-complete.

Nevertheless, in many cases, the enormous complexity upper bounds provided by the Length Function Theorems are matched by similar lower bounds on the complexity of the Coverability and Termination *problems*, for instance for reset/transfer Petri nets [$\mathbf{F}_\omega$-complete, see 42], lossy channel systems [$\mathbf{F}_{\omega^\omega}$-complete, see 13], timed-arc Petri nets [$\mathbf{F}_{\omega^\omega\omega}$-complete, see 27], or priority channel systems [$\mathbf{F}_{\varepsilon_0}$-complete, see 26].

Our goal in this section is to present the common principles behind these $\mathbf{F}_\alpha$-hardness proofs. We will avoid most of the technical details, relying rather on simple examples to convey the main points: the interested readers will find all details in the references.

Let us consider a WSTS model like broacast protocols or lossy channel systems. Without a rich repertoire of $\mathbf{F}_\alpha$-complete problems, one proves $\mathbf{F}_\alpha$-hardness by reducing, e.g., from the acceptance problem for a H^{ω^α}-space bounded Minsky machine M. In order to simulate M in the WSTS model at hand, the essential part is to design a way to compute $H^{\omega^\alpha}(n_0)$ reliably and store this number as a WSTS state, where it can be used as a working space for the simulation of M. In all the cases we know, these computations cannot be performed directly (indeed, our WSTS are not Turing-powerful), but $\mathcal{S}_M$, the constructed WSTS, is able to *weakly* compute such values. This means that $\mathcal{S}_M$ may produce the correct value for $H^{\omega^\alpha}(n_0)$ but also (nondeterministically) some smaller values. However, the reduction is able to include a check that the computation was actually correct, either at the end of the simulation [42, 13, 27, 26]—by weakly computing the inverse of H^{ω^α} and testing through the coverability condition whether the final configuration is the one we started with—, or continuously at every step of the simulation [33].

5.1 Hardy Computations

As an example, when $\alpha < \omega^k$, one may weakly compute H^α and its inverse using a broadcast protocol with $k + O(1)$ locations. In order to represent an ordinal $\alpha = \omega^{k-1} \cdot c_{k-1} + \cdots + \omega^0 \cdot c_0$ in CNF, one can employ a configuration $s_\alpha = \{p_0^{c_0}, \ldots, p_{k-1}^{c_{k-1}}\}$ in a broadcast protocol having locations $p_0, \ldots, p_{k-1}$ among others.

There remains other issues with ordinal representations in a WSTS state (see Sec. 5.2), but let us first turn to the question of computing some $H^\alpha(n)$. The definition of the Hardy functions is based on very fine-grained steps, and this usually simplifies their implementations. We can reformulate (4) as a rewrite system over pairs (α, x) of an ordinal and an argument:

$$(\alpha + 1, x) \to (\alpha, x + 1)\,, \qquad (\lambda, x) \to (\lambda(x), x)\,. \tag{4'}$$

A sequence $(\alpha_0, x_0) \to (\alpha_1, x_1) \to \cdots \to (\alpha_\ell, x_\ell)$ of such "Hardy steps" implements (4) and maintains $H^{\alpha_i}(x_i)$ invariant. It must terminates since $\alpha_0 > \alpha_1 > \cdots$ is decreasing. When eventually $\alpha_\ell = 0$, the computation is over and the result is $x_\ell = H^{\alpha_0}(x_0)$.

Example 5.1 (Hardy Computations in Broadcast Protocols). Implementing (4') in a broadcast protocol requires us to consider two cases. Recognizing whether α_i is a successor boils down to checking that $c_0 > 0$ in the CNF. In that case, and assuming the above representation, (4') is implemented by moving one process from location p_0 to a location x where the current value of x_i is stored. The broadcast protocol will need a rule like $p_0 \xrightarrow{\mathrm{sp}(x)} \bot$.

Alternatively, α_i is a limit $\gamma + \omega^b$ when $c_0 = c_1 = \ldots = c_{b-1} = 0 < c_b$. In that case, (4') is implemented by moving one process out of the p_b location, and adding to p_{b-1} as many processes as there are currently in x. This can be implemented by moving temporarily all processes in x to some auxiliary x_{tmp} location, then putting them back in x one by one, each time spawning a new process in p_{b-1}.

The difficulty with these steps (and with recognizing that α_i is a limit) is that one needs to test that some locations are empty, an operation not provided in broadcast protocols (adding emptiness tests would make broadcast protocols Turing-powerful, and would break the monotonicity of behaviour). Instead of being tested, these locations can be forcefully emptied through broadcast steps. This is where the computation of $H^{\alpha_i}(x_i)$ may err and end up with a smaller value, if the locations were not empty. We refer to [42] or [41, Chap. 3] for a detailed implementation of this scheme using lossy counters machines: the encoding therein can easily be reformulated as a broadcast protocol:

Fact 5.2. *Termination and Coverability of broadcast protocols are* $\mathbf{F}_\omega$*-hard.*

5.2 Robust Encodings

The above scheme for transforming pairs (α, x) according to Eq. (4') can be used with ordinals higher than ω^k. Ordinals up to ω^{ω^ω} have been encoded as

configurations of lossy channel systems [13], of timed-arc nets (up to $\omega^{\omega^{\omega^{\omega}}}$, see [27]), and of priority channel systems (up to ε_0, see [26]). The operations one performs on these encodings are recognizing whether an ordinal is a successor or a limit, transforming an $\alpha + 1$ in α, and a λ in $\lambda(x)$. Such operations can be involved, depending on the encoding and the facilities offered by the WSTS: see [27] for an especially involved example.

It is usually not possible to perform Hardy steps exactly in the WSTS under consideration. Hence one is content with weak implementations that may err when realizing a step $(\alpha_i, x_i) \to (\alpha_{i+1}, x_{i+1})$. One important difficulty arises here: it is not enough to guarantee that any weak step $(\alpha_i, x_i) \to (\alpha', x')$ has $\alpha' \leq \alpha_{i+1}$ and $x \leq x_{i+1}$. One further needs $H^{\alpha'}(x') \leq H^{\alpha_{i+1}}(x_{i+1})$, a property called "robustness". Since Hardy functions are in general not monotone in the α exponent (see [41]), extra care is needed in order to control what kinds of errors are acceptable when ending up with (α', x') instead of (α_{i+1}, x_{i+1}). We invite the reader to have a look at the three above-mentioned papers for examples of how these issues can be solved in each specific case.

6 Concluding Remarks

As the claim *Well-Structured Transition Systems Everywhere!* made in the title of [24] has been further justified by twelve years of applications of WSTS in various fields, the need to better understand the computational power of these systems has also risen. This research program is still very new, but it has already contributed mathematical tools and methodological guidelines for

- proving upper bounds, based on *length functions theorems* that provide bounds on the length of controlled bad sequences. We illustrated this on two algorithms for Coverability and Termination in Sec. 4, but the same ideas are readily applicable to many algorithms that rely on a wqo for their termination—and thus not only in a WSTS context—: one merely has to find out how the bad sequences constructed by the algorithm are *controlled*.
- establishing matching lower bounds: here our hope is for the problems we have proven hard for some complexity class $\mathbf{F}_\alpha$ to be reused as convenient "master" problems in reductions. Failing that, such lower bound proofs can also rely on a reusable framework developed in Sec. 5: our reductions from Turing or Minsky machines with bounded resources construct the machine workspace as the result of a Hardy computation, thanks to a suitable robust encoding of ordinals.

There are still many open issues that need to be addressed to advance this program: to develop length function theorems for more wqos, to investigate different wqo algorithms (like the computation of upward-closed sets from oracles for membership and vacuity by Goubault-Larrecq [25]), and to populate the catalog of master $\mathbf{F}_\alpha$-hard problems, so that hardness proofs do not have to proceed from first principles and can instead rely on simpler reductions.

We hope that this paper can be used as an enticing primer for researchers who have been using WSTS as a decidability tool only, and are now ready to use them for more precise complexity analyses.

Acknowledgments. We thank Christoph Haase and Prateek Karandikar for their helpful comments on an earlier version of this paper. The remaining errors are of course entirely ours.

References

1. Abdulla, P.A.: Well (and better) quasi-ordered transition systems. Bull. Symbolic Logic 16(4), 457–515 (2010)
2. Abdulla, P.A., Čerāns, K., Jonsson, B., Tsay, Y.K.: Algorithmic analysis of programs with well quasi-ordered domains. Inform. and Comput. 160(1/2), 109–127 (2000)
3. Abdulla, P.A., Delzanno, G., Van Begin, L.: A classification of the expressive power of well-structured transition systems. Inform. and Comput. 209(3), 248–279 (2011)
4. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. Inform. and Comput. 127(2), 91–101 (1996)
5. Abriola, S., Figueira, S., Senno, G.: Linearizing bad sequences: upper bounds for the product and majoring well quasi-orders. In: Ong, L., de Queiroz, R. (eds.) WoLLIC 2012. LNCS, vol. 7456, pp. 110–126. Springer, Heidelberg (2012)
6. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: POPL 2010, pp. 7–18. ACM (2010)
7. Barceló, P., Figueira, D., Libkin, L.: Graph logics with rational relations and the generalized intersection problem. In: LICS 2012, pp. 115–124. IEEE Press (2012)
8. Bertrand, N., Schnoebelen, P.: Computable fixpoints in well-structured symbolic model checking. Form. Methods in Syst. Des. (to appear, 2013)
9. Bozzelli, L., Ganty, P.: Complexity analysis of the backward coverability algorithm for VASS. In: Delzanno, G., Potapov, I. (eds.) RP 2011. LNCS, vol. 6945, pp. 96–109. Springer, Heidelberg (2011)
10. Bresolin, D., Della Monica, D., Montanari, A., Sala, P., Sciavicco, G.: Interval temporal logics over finite linear orders: The complete picture. In: ECAI 2012. Frontiers in Artificial Intelligence and Applications, vol. 242, pp. 199–204. IOS (2012)
11. Cardoza, E., Lipton, R., Meyer, A.R.: Exponential space complete problems for Petri nets and commutative subgroups. In: STOC 1976, pp. 50–54. ACM (1976)
12. Chambart, P., Schnoebelen, P.: Post embedding problem is not primitive recursive, with applications to channel systems. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 265–276. Springer, Heidelberg (2007)
13. Chambart, P., Schnoebelen, P.: The ordinal recursive complexity of lossy channel systems. In: LICS 2008, pp. 205–216. IEEE Press (2008)
14. Cichoń, E.A., Tahhan Bittar, E.: Ordinal recursive bounds for Higman's Theorem. Theor. Comput. Sci. 201(1-2), 63–84 (1998)
15. Clote, P.: On the finite containment problem for Petri nets. Theor. Comput. Sci. 43, 99–105 (1986)
16. Emerson, E.A., Namjoshi, K.S.: On model checking for non-deterministic infinite-state systems. In: LICS 1998, pp. 70–80. IEEE Press (1998)

17. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: LICS 1999, pp. 352–359. IEEE Press (1999)
18. Esparza, J., Ganty, P., Majumdar, R.: Parameterized verification of asynchronous shared-memory systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 124–140. Springer, Heidelberg (2013)
19. Figueira, D.: Alternating register automata on finite words and trees. Logic. Meth. in Comput. Sci. 8(1), 22 (2012)
20. Figueira, D., Figueira, S., Schmitz, S., Schnoebelen, P.: Ackermannian and primitive-recursive bounds with Dickson's Lemma. In: LICS 2011, pp. 269–278. IEEE Press (2011)
21. Finkel, A.: A generalization of the procedure of Karp and Miller to well structured transition systems. In: Ottmann, T. (ed.) ICALP 1987. LNCS, vol. 267, pp. 499–508. Springer, Heidelberg (1987)
22. Finkel, A.: Decidability of the termination problem for completely specificied protocols. Distributed Computing 7(3), 129–135 (1994)
23. Finkel, A., Goubault-Larrecq, J.: Forward analysis for WSTS, part II: Complete WSTS. Logic. Meth. in Comput. Sci. 8(4:28) (2012)
24. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. 256(1-2), 63–92 (2001)
25. Goubault-Larrecq, J.: On a generalization of a result by Valk and Jantzen. Research Report LSV-09-09, Laboratoire Spécification et Vérification, ENS Cachan, France (2009)
26. Schmitz, S., Schnoebelen, P.: The power of well-structured systems. In: D'Argenio, P.R., Melgratti, H. (eds.) CONCUR 2013. LNCS, vol. 8052, pp. 5–24. Springer, Heidelberg (2013)
27. Haddad, S., Schmitz, S., Schnoebelen, P.: The ordinal-recursive complexity of timed-arc Petri nets, data nets, and other enriched nets. In: LICS 2012, pp. 355–364. IEEE Press (2012)
28. Jančar, P.: Nonprimitive recursive complexity and undecidability for Petri net equivalences. Theor. Comput. Sci. 256(1-2), 23–30 (2001)
29. Jurdziński, M., Lazić, R.: Alternating automata on data trees and XPath satisfiability. ACM Trans. Comput. Logic 12(3) (2011)
30. Kruskal, J.B.: The theory of well-quasi-ordering: A frequently discovered concept. J. Comb. Theory A 13(3), 297–305 (1972)
31. Kurucz, A.: Combining modal logics. In: Handbook of Modal Logics, ch. 15, pp. 869–926. Elsevier Science (2006)
32. Lasota, S., Walukiewicz, I.: Alternating timed automata. ACM Trans. Comput. Logic 9(2), 10 (2008)
33. Lazić, R., Ouaknine, J., Worrell, J.: Zeno, Hercules and the Hydra: Downward rational termination is Ackermannian. In: MFCS 2013. LNCS. Springer (to appear, 2013)
34. Löb, M., Wainer, S.: Hierarchies of number theoretic functions. I. Arch. Math. Logic 13, 39–51 (1970)
35. Mayr, E.W., Meyer, A.R.: The complexity of the finite containment problem for Petri nets. J. ACM 28(3), 561–576 (1981)
36. McAloon, K.: Petri nets and large finite sets. Theor. Comput. Sci. 32(1-2), 173–183 (1984)
37. Milner, R.: Operational and algebraic semantics of concurrent processes. In: Handbook of Theoretical Computer Science, ch. 19, pp. 1201–1242. Elsevier Science (1990)

38. Ouaknine, J., Worrell, J.: On the decidability and complexity of Metric Temporal Logic over finite words. Logic. Meth. in Comput. Sci. 3(1), 8 (2007)
39. Rackoff, C.: The covering and boundedness problems for vector addition systems. Theor. Comput. Sci. 6(2), 223–231 (1978)
40. Schmitz, S., Schnoebelen, P.: Multiply-recursive upper bounds with Higman's Lemma. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 441–452. Springer, Heidelberg (2011)
41. Schmitz, S., Schnoebelen, P.: Algorithmic aspects of wqo theory. Lecture notes (2012), http://cel.archives-ouvertes.fr/cel-00727025
42. Schnoebelen, P.: Revisiting Ackermann-hardness for lossy counter machines and reset Petri nets. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 616–628. Springer, Heidelberg (2010)
43. Schwichtenberg, H., Wainer, S.S.: Proofs and Computation. Perspectives in Logic. Cambridge University Press (2012)
44. Urquhart, A.: The complexity of decision procedures in relevance logic II. J. Symb. Log. 64(4), 1774–1802 (1999)
45. Weiermann, A.: Complexity bounds for some finite forms of Kruskal's Theorem. J. Symb. Comput. 18(5), 463–488 (1994)

Impact of Resource Sharing on Performance and Performance Prediction: A Survey

Andreas Abel, Florian Benz, Johannes Doerfert, Barbara Dörr, Sebastian Hahn, Florian Haupenthal, Michael Jacobs, Amir H. Moin, Jan Reineke, Bernhard Schommer, and Reinhard Wilhelm

Saarland University, Saarbrücken, Germany

Abstract Multi-core processors are increasingly considered as execution platforms for embedded systems because of their good performance/energy ratio. However, the interference on shared resources poses several problems. It may severely reduce the performance of tasks executed on the cores, and it increases the complexity of timing analysis and/or decreases the precision of its results. In this paper, we survey recent work on the impact of shared buses, caches, and other resources on performance and performance prediction.

1 Introduction

Multi-core processors are increasingly considered as execution platforms for embedded systems since they offer a good energy-performance tradeoff and seem to support transitions from federated to integrated system architectures in the automotive and avionics domains. Many applications implemented on such multi-core platforms are safety- and some also time-critical. A critical issue is the reduced predictability of such systems resulting from the interference of different applications on shared resources. These interferences can be at least of two kinds: Several applications may request a resource at the same time, but the resource can only admit one access at a time. As a consequence, an arbitration mechanism may delay the request of all but one application, thus slowing down the other applications. This is the case of resources like buses, typically called *bandwidth resources*. On the other hand, one application may also change the state of a shared resource such that another application using that resource will suffer from a slowdown. This is the case with shared caches, which fall into the class of *storage resources*. Most of the treatments of the interferences on shared resources found in the literature consider the detrimental effect of interferences. In the case of shared caches, however, the interference of one application A_1 on another co-running application A_2 could even speed up A_2 if A_1 would perform the right cache prefetching for A_2.

Interference on shared resources makes worst-case execution time (WCET) analysis of applications more difficult since a task or a thread can no longer be analyzed for its timing behavior in isolation. All potential interferences slowing down (or speeding up) the task under analysis have to be considered. This leads to a combinatorial explosion of the analysis complexity, as all possible interleavings of different threads have to be analyzed. For that reason, currently, no sound timing-analysis method for multi-core platforms with shared resources exists.

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 25–43, 2013.

This survey considers several aspects of the execution of sets of tasks on multi-core platform that have to do with the interference of the tasks on shared resources. One question is how the actual performance of tasks is slowed down by other co-running tasks. The other is how to compute bounds on the slow-down in order to derive guarantees for the timing behavior. A major problem is the increased complexity of this task compared to the single-task single-core case.

Caches are a particular case of *storage resources*. Several approaches exist for the treatment of shared caches in attempts to derive timing guarantees. Cache partitioning eliminates the interference between tasks. Static analysis of non-partitioned shared caches attempts to safely bound the interference. The definite comparison between these two approaches has yet to be done.

Buses are instances of *bandwidth resources*. Several protocols exist for the arbitration of shared buses, which can be classified as either time-driven, event-driven, or hybrid combinations of both. Static analysis can be used to determine good slot assignments in time-driven protocols like TDMA, and it can be used to determine bounds on the access delays in event-driven state-based protocols like FCFS and round robin.

1.1 Ways to Derive Guarantees

In order to guarantee the timeliness of tasks in a hard real-time system, one needs upper bounds on the execution times of the tasks.

As long as a task executes in isolation on a multi-core system (without co-running tasks), existing techniques for timing estimation could be applied. In case of parallel workloads (with co-running tasks), a sound approach for timing analysis of multi-core systems has to take into account the interferences, as described in detail in Sections 2 and 3.

Approaches to determine upper bounds on execution times of tasks on multi-core processors can be classified into two groups:

- Approaches achieving *performance isolation* by hardware and/or software techniques, e.g. by employing TDMA arbitration of busses. Performance isolation implies *timing composability* and permits the use of standard single-core timing analysis techniques with minor modifications. While this makes timing analysis comparatively easy, the challenge in such approaches is to make efficient use of shared resources by partitioning them appropriately for the given workload.
- Approaches analyzing the mutual effects of co-running tasks on each other's execution time. Such approaches require new timing analysis techniques that differ greatly from those employed in the single-core single-task case.

Different methods have been proposed or are pursued to derive guarantees for the timeliness of sets of tasks in a parallel workload setting when performance isolation is not given. First, there is the classification according to whether the software is analyzed or executed.

- The *static analysis* of a whole set of concurrently executed applications may deliver a sound and precise guarantee for the timing behavior. The problem is the huge complexity of this approach.
- *Measurement-based* methods are in general not able to derive guarantees, neither in the single-core nor in the multi-core case.

The particular contribution to the execution-time bounds of the interference on shared resources can be dealt with in different ways:

- The *Murphy* approach assumes maximal interference on each access to a shared resource [1]. This assumption can be easily integrated into existing single-core timing analysis techniques. The *Murphy* approach will clearly give sound, but the most pessimistic execution-time bounds.
- The *slowdown factor* approach attempts to explicitly quantify the worst-case impact of the interferences in shared resources on the timing of a task caused by co-running tasks. The obtained slowdown factor can then be used to obtain an estimate on the execution times of a task in a parallel workload from an estimate in the isolated case. Existing approaches aim at quantifying the slowdown of a task in the worst case by measurement-based techniques. These measurement-based approaches employ so-called *resource-stressing benchmarks*, which are constructed for particular resources to produce the maximal slowdown on co-running tasks due to conflicts on this resource. In Section 4, we will see that attempts in this direction may be both unsound and overly pessimistic. Resource-stressing benchmarks are, in general, independent of the application that is slowed down by co-running these benchmarks. Therefore, one single resource-stressing benchmark can hardly slow down the application in the worst way. The fact that resource-stressing benchmarks might not be sound is demonstrated by an imaginary application-specific *worst companion* (see Section 4).
- Finally, the static analysis of a whole set of concurrently executed applications may deliver a sound and precise guarantee for the timing behavior. The problem is the huge complexity of this approach. To reduce analysis complexity, existing static analysis approaches separate analysis into two phases: The first phase determines a bound on the execution time of each task in isolation and a characterization of its resource-access behavior. The second analysis phase then uses this characterization to bound the impact of interference on the execution times of all tasks. The sum of the two bounds for each task then yields an estimate of the task's worst-case execution time. Such approaches are discussed in Section 2. For soundness, all of these approaches rely on *timing compositional* hardware architectures [2], which permit to account for the cost of interference in such a compositional way. Unfortunately, many existing hardware architectures exhibit domino effects and timing anomalies and are thus out of the scope of such an approach.

1.2 Terminology

We will need a few terms for this survey. The *(individual) access delay* is the interval between the time of an individual access request and the time when

this request is granted. This can sometimes be determined by measurement. The *worst-case access delay* is the access delay in a worst-case scenario. It is typically derived from the arbitration protocol and some system parameters, e.g. the number of concurrently executed tasks and the slot size in bus protocols. The *overall access delay* is the sum of all the access delays encountered during a task's execution. An upper bound on the overall access delay is sometimes used in combination with a WCET estimate to compute a worst-case response time (WCRT) estimate.

2 Bandwidth Resources, in Particular Shared Buses

In computer architectures, buses are used to transfer data between different components of a computer. The components allowed to access a particular bus are called the *accessors* of the bus. In order to reduce cost and complexity of the overall system, often only one accessor is allowed to access the bus at a time. In this case, the limited bandwidth of such a *shared bus* is shared solely along the dimension of time. We only consider such buses in the following.

Several accessors may attempt to access the bus at the same time and thus cause a *bus conflict*. Arbitration mechanisms resolve bus conflicts by only granting access to one accessor at a time. There are three main classes of resource arbitration mechanisms. *Time-driven arbitration* uses a predefined bus schedule, which assigns time slots of fixed size to particular accessors. This is commonly referred to as *TDMA* (Time Division Multiple Access). *Event-driven arbitration* mechanisms decide at runtime, which accessor is granted resource access next. These decisions usually depend on the access histories of all accessors. Prominent examples are *Round Robin* or *FCFS* (First-Come-First-Serve) resource arbitrations. *Hybrid arbitration* mechanisms split their arbitration period into segments of fixed length. Static segments use time-driven resource arbitration and dynamic segments use event-driven resource arbitration. A member of this class is the *FlexRay* [3] bus protocol used in the automotive industry [4,5].

One can distinguish between *synchronous* and *asynchronous* bus accesses. Requests for *asynchronous access* to a shared bus can be buffered until they are finally granted by the arbiter. This way, the requesting application can immediately continue its execution. Applications that only rely on asynchronous resource accesses can achieve the same performance as in combination with a dedicated bus provided sufficient buffer sizes, enough bandwidth, and sufficiently delayed accesses to the results. In contrast, requests for *synchronous access*, e.g. memory load requests, block the requesting application until they are granted by the arbiter. The blocking of requests for synchronous bus accesses leads to additional stalls of the requesting processing unit. Such bus interference effects may decrease the performance of a processing unit compared to a dedicated bus scenario on average as well as in the worst case.

In a multiprocessing environment, a processing unit may access a shared bus *explicitly* or *implicitly*. Explicit accesses are performed by commands occurring in the program, and can therefore be more easily analyzed than implicit accesses. A first-level cache, private to a processing unit, for example, accesses a shared memory bus for cache reloads. The cache-coherence protocol for private caches also accesses the shared bus. Implicit accesses are harder to track by static analysis than explicit accesses since they are subject to more uncertainty.

2.1 Analysis Approaches and Task Model

Naive bounds on the WCET and WCRT of tasks assume bus access requests to always be granted immediately. They are obviously not sound in the presence of bus interferences. A common approach is to first derive naive bounds on the WCET and to account for possible interference effects in an additional analysis step focussed on the bus interference.

Literature on worst-case timing analysis for systems with shared bus resources and synchronous bus accesses is mainly concerned with this additional analysis step. The common assumption is that adding the access delays to the naive timing bounds will lead to sound WCET bounds. This can only be guaranteed for compositional timing models of hardware platforms [2].

To simplify the analysis of bus interference, all surveyed approaches adopt a *task model* based on so-called superblocks. Each task is described as a sequence of superblocks (see Figure 1). Upper bounds on the amount of computation time ($exec_i$) and the number of resource accesses (μ_i) per superblock (s_i) serve as input and are used as common abstraction of tasks [6]. The bounds on the amount of computation time are assumed to be the result of a naive WCET analysis ignoring completely the time needed to access the bus. For simplicity, our figures assume that a bus access always takes one time unit to be served once it is granted. In the presence of implicit bus accesses it can be a major challenge to derive tight bounds on the number of resource accesses of a superblock. This is an open research problem.

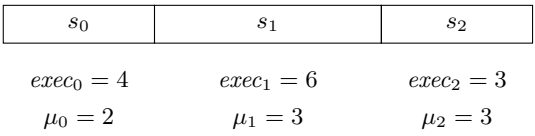


Fig. 1. A task consisting of three superblocks

The approaches in the literature differ in the class of resource arbitration considered. While Pellizzoni et al. [7,8] are concerned with event-driven resource arbitration mechanisms, Schranzhofer et al. [6] treat TDMA resource arbitration. The determination of safe timing bounds in the presence of a shared bus and a hybrid arbitration mechanisms is described in Schranzhofer et al. [9].

Time-driven Resource Arbitration. An inherent property of time-driven resource arbitration is that the overall access delay experienced by a task on one processing unit is independent of the access requests issued by concurrent accessors. Thus, current approaches for worst-case timing analysis of systems with shared TDMA buses bound the overall access delay of a given task by considering its bus accesses and their possible distributions across a known bus schedule [6]. Optimization approaches try to find a pair of task assignment and bus schedule leading to a low WCRT bound [10].

The overall access delay of the processor under consideration (PuC) is maximized when it is blocked as long as possible by the bus arbitration. The PuC is blocked whenever it tries to access the bus in a time slot assigned to another accessor. Furthermore, it is blocked in a time slot assigned to itself if its access request is released too late to be completed in the current time slot.

The amount of overall blocking experienced by a superblock depends on how its bus accesses and its computation time are distributed within the superblock. Possible distributions are restricted by the given upper bounds on the number of bus accesses and the computation time per superblock. Intuitively, a bus access released early enough in a time slot assigned to the PuC does not lead to any blocking of the PuC. In order to maximize the overall blocking, a distribution of accesses and computation time in the superblock should avoid such access releases as far as possible.

Algorithms to calculate a WCRT bound for a superblock run through a sequence of points in time starting with the worst-case release time of the superblock. For each point in time, they decide if a bus access or a certain amount of computation time could lead to a globally maximized blocking. This decision determines which point in time has to be considered next. Uncertainty in this decision may lead to case splits. Termination is guaranteed as the maximal amount of resource accesses and computation time of a superblock will be consumed at some point in time. Such algorithms construct a distribution of bus accesses and computation time leading to a tight bound on the WCRT. To avoid case distinction and thereby reduce the complexity of the problem, it is possible to use coarse under- and overapproximations as intermediate result. This interval can be further refined using a binary search that excludes candidates guaranteed to be infeasible [6].

A superblock with its upper bounds on the computation time and the number of bus accesses is an abstraction of a fragment of a task. Obviously, these upper bounds may allow for different distributions of bus accesses and computation time across the superblock than those found in the original task. As a consequence, the worst-case distribution of accesses determined by an algorithm may actually be infeasible. The amount of pessimism introduced by the superblock abstraction remains to be evaluated.

In order to decrease the upper bounds on the WCRT of a task, it is beneficial to split it up into superblocks in a way that each superblock either only performs computations or only performs bus accesses. This is referred to as *dedicated access model* [6]. A deterministic execution model for time-critical systems

introduced by Boniol et al. [11] distinguishes between execution and communication slices in a similar way. The reason for the observed improvement is that the bounds on the WCRT of a superblock are the consequence of very unbeneficial interleavings of its bus accesses and its computation time with respect to the bus schedule's time slots.

However, it is not clear at which granularity superblocks should be abstracted from a task's code. Furthermore, it remains unclear which restrictions a task's code has to fulfill in order to be abstracted to superblocks following the dedicated access model. A simple possibility is to enforce a strict separation of computation and bus accesses by the programmer. This is, however, not possible in the presence of implicit bus accesses, which are not visible to the programmer and have to be detected by a consecutive static analysis.

Event-Driven Resource Arbitration. It is a common assumption in the literature that the latency of an individual resource access of the PuC can be bounded. Either an arbitration protocol provides this property to all accessors, or at least to the PuC. In the presence of event-driven resource arbitration, there are two approaches to bound the effect of interference, discussed in more detail below:

1. By taking into account the arbitration logic.
2. By taking into account the amount of competing resource accesses.

In both approaches, the common principle is to construct the worst-case scenario, that is the scenario that maximizes the latency of a request based on the knowledge about the interference.

Knowledge about the maximum number of bus accesses in a superblock of the PuC allows to bound its overall access delay: For round-robin arbitration, the worst-case scenario is that all other processing units are allowed to perform one access before the PuC is allowed to do so [7]. In FCFS arbitration, an interference bound has to additionally take into account the maximum number of access requests that can be released at the same time by concurrent processing units if this number is greater than one [8]. We call this *delay bounding based on the arbitration logic*. Of course these bounds are only tight provided that the concurrent processing units can really issue this number of bus requests while the superblock is executed. This remark leads us to the second bounding factor on the interference.

It is an inherent property of event-driven arbitration mechanisms, that a processing unit can only be blocked if and when another processing unit is requesting bus access. Furthermore, many event-driven arbitration mechanisms do not allow to interrupt and restart a bus access once it is granted. Provided that these two properties hold, a superblock on the PuC cannot experience more overall access delay than the time needed to perform all the accesses concurrent accessors might release while the superblock is executed [7,8]. We refer to this as *delay bounding based on the amount of concurrently requested access time*.

In order to bound the amount of concurrently requested access time for a given superblock, all possible interleavings of its bus accesses with concurrent

access sequences have to be taken into account. As this may be computationally infeasible in general, arrival curves [12,13] are introduced as an abstraction of the concurrent access behavior. They bound the maximal amount of access time that concurrent accessors might request in a time interval of given length. It is possible to obtain them from the superblock abstractions of the task on concurrent processing units [8]. Let function $R(t)$ describe the cumulative amount of access time requested by a concurrent accessor until time t. An arrival curve $\alpha(\Delta)$ bounds the amount of access time that can be requested in any time interval of length Δ:

$$\forall t_1 : \ \forall t_0 \leq t_1 : \ R(t_1) - R(t_0) \leq \alpha(t_1 - t_0)$$

The use of arrival curves for more than one concurrent accessor inherently over-approximates the experienced blocking. A longer running superblock can potentially suffer from more delay caused by a particular concurrent accessor than a quickly executed superblock. Therefore, the involved fixed point iteration has to pessimistically assume for each concurrent accessor that it can already profit from the delay caused by all other concurrent accessors. Yet, this loss of precision is bearable regarding the complexity of considering all possible interleavings of bus access sequences of the concurrent accessors.

In addition, arrival curves also incorporate concurrent access sequences that can possibly never be executed in parallel with a particular superblock. Therefore, arrival curves may introduce additional pessimism with respect to the concurrent access request behavior if that behavior exhibits a heterogenous structure along the dimension of time.

Hybrid Resource Arbitration. For hybrid resource arbitration, it is more challenging to find a distribution pattern of bus accesses and computation time across the bus arbitration segments that leads to a globally worst response time. When constructing worst-case scenarios, the static and dynamic segments need to be taken into consideration jointly.

While it would be feasible to derive worst-case scenarios considering only the static or only the dynamic segments with the methods described above, the derived bounds would likely be very pessimistic.

To avoid high over-estimation, a dynamic programming approach presented by Schranzhofer et al. [9] enumerates all possible distributions of accesses in a superblock and interfering accesses of concurrent processing units across the segments of the bus arbitration.

2.2 Summary of the State of the Art and Open Problems

It is possible to determine safe bounds on the WCRTs of tasks for systems with synchronous accesses to a shared bus. Upper bounds on the naive WCETs and the number of bus accesses for each superblock are assumed as input. Approaches described in the literature treat different classes of bus-arbitration mechanisms.

Each of the approaches is backed by its own separate, rigorous formalism. It would be beneficial to identify more generic approaches to reason about all classes of arbitration mechanisms treated so far as well as possible future ones.

Furthermore, the current level of abstraction (superblock-based task model, arrival curves) is quite high and possibly results in severe overestimation of the actual interference. It would be valuable to examine the tradeoff between analysis efficiency and precision by studying more precise task characterizations.

All presented approaches require the bounds on the amount of computation time and the access times and delays to be compositional. Therefore most authors assume a timing-compositional hardware platform [2]. Such a platform allows for a precise description of its temporal behavior by a compositional timing model. Future work should consider the treatment of hardware platforms that currently only allow for a precise timing analysis in combination with models exhibiting timing anomalies.

3 Storage Resources, in Particular Shared Caches

Caches are used to bridge the large latency gap between modern processor pipelines and main memory. They are small, low-latency on-chip memories that buffer a subset of the contents of the main memory. Cache replacement logic decides at runtime which memory blocks to store in the cache. Sequential programs running on single-core machines usually exhibit high *spatial* and *temporal locality* and thus experience a high cache-hit ratio. As a consequence, the average memory access latency is close to that of the cache in such a scenario. Precise and efficient static analysis of the behavior of a private cache is possible if the memory accesses of the application and the replacement logic are predictable [14].

Current and upcoming multi-core processors feature *private* first- and sometimes second-level caches and *shared* higher-level caches. In theory, large shared caches promise a more efficient use of expensive die area than an array of small private caches of the same aggregate capacity: their capacity can be shared flexibly according to the needs of the programs running on the connected cores. In addition, if applications running on different cores share data, they can communicate more efficiently through the shared cache than through main memory.

Unfortunately, the behavior of current shared caches is hard to predict statically. The reason for their unpredictability is the interference between accesses originating from different cores. As with buses, caches cannot serve multiple memory accesses completely in parallel. Pipelining of accesses is possible, but some "bandwidth interference" remains. However, the main challenge with shared caches is that the state of the cache depends on the precise interleaving of accesses from multiple cores. Accesses from different cores are typically served on a first-come first-served basis. Their interleaving thus depends on the relative execution speeds of the applications running on these cores, which—among other things—depend on their cache performance, which in turn depends on the cache state. This cyclic dependency between the interleaving of the accesses and the cache state makes precise and efficient analysis hard or even impossible

in general. Additional complications—not detailed in this survey—arise from coherence protocols, which need to be employed when applications running on different cores share memory. Now, even if a shared-cache analysis taking into account the interactions of multiple applications would exist, such an analysis would have to have precise knowledge of all applications, their mapping, and their scheduling. This would render independent and incremental certification impossible.

Maybe unsurprisingly, it has been observed that uncontrolled sharing of caches that does not take into account that cached memory blocks belong to different cores or applications is also detrimental from an average-case performance perspective. For example, an application with low temporal and spatial locality that generates many first-level cache misses can acquire a large portion of the shared cache without any benefit to system performance.

A common approach found in recent literature is to take into account the core or application a cache block belongs to when deciding which block to replace. Typically, the cache is conceptually partitioned among the cores, so that memory blocks compete for cache space only with other memory blocks of the same core. Within each partition a common replacement policy such as least-recently used is applied. The partition sizes are chosen with different objectives in mind, such as aggregate performance and fairness. In addition or even instead of varying partition sizes, some approaches also adapt the schedule of applications with these objectives in mind. Both decisions about scheduling and partition sizes can happen either statically or dynamically and are taken based on a static or dynamic characterization of the different core's memory access behavior.

3.1 Cache Partitioning

In the following, we provide a non-exhaustive survey of the vast existing literature on

- methods to partition caches,
- approaches to determine good partition sizes, and
- methods to schedule tasks taking into account a shared cache.

How to Partition a Cache.
There are various approaches to partition caches. They can be distinguished by

- the partitioning scheme: set-based [15,16] or way-based [17,18], and by
- their implementation: in software [15,16] or in hardware [19,17,18].

In set-based partitioning, each core is given exclusive access to a subset of all cache sets. This can be realized both in hardware and in software. Hardware-based solutions can realize set-based partitioning by adapting the mapping of memory blocks to cache sets depending on the core that issued the memory access. If virtual memory is employed and the shared cache is physically-indexed, set-based partitioning can also be realized in software by *page coloring* [20]: In page coloring, physical pages mapping to the same set of cache sets are assigned the same *color*. Then, the set of colors is partitioned among the different

processes. Virtual pages belonging to a particular process are only mapped to physical pages of the colors assigned to that process.

In way-based partitioning, the cache is partitioned along the cache ways: each core is given exclusive access to a subset of all cache ways. This can only be realized in hardware, by accordingly adapting the cache replacement logic.

Both schemes have their advantages and drawbacks: In particular in low-associativity caches, way-based partitioning only allows for a coarse-grained allocation of the cache space. Set-based partitioning, in particular its hardware-based variant, allows for a more fine-grained allocation of the cache space as the number of ways usually exceeds a cache's associativity. The main drawback of set-based partitioning is that changing partition sizes is expensive. Changing the coloring of pages or changing the mapping of memory blocks to cache sets implies that the cached data needs to either be invalidated or moved to their new locations in the cache. As a consequence, hardware-based solutions usually rely on way-based partitioning, in which changing partition sizes is cheap. Software-based solutions are forced to rely on set-based partitioning.

How to Determine Sizes of Partitions. The choice of partition sizes has a strong effect on the performance of each corunning task, and thus the system performance. Multiple metrics have been proposed to measure system performance, the most common metric being *throughput*, i.e. the sum of the IPCs (instructions per cycle) of all corunning tasks. Other metrics combine performance with fairness goals.

To arrive at a good partition in terms of a particular performance metric, the effect of a partition size on each corunning task needs to be characterized in some form. The following characterizations have been used to drive the partitioning choice:

- *Miss ratio curves* [15] capture the miss ratio of a task in terms of the task's cache partition size. Unfortunately, miss ratios, i.e., ratios between cache misses and memory accesses, are not directly correlated with execution times. A high miss ratio does not imply that the task spends much time waiting for memory accesses. Thus, minimizing miss ratios does not necessarily maximize system throughput. A related metric proposed by Tam et al. [21] are stall-rate curves, which capture the share of overall execution time a task is stalled for memory.
- *Misses per 1000 instruction (MPKI) in terms of cache size* [18]. This metric is much more closely tied to the cache's impact on execution times. Still, it may not be perfectly correlated with the resulting performance in terms of cycles per instruction.
- Cycles per instruction (CPI) in terms of cache size [18]. This would be the ideal metric when optimizing throughput. However, in contrast to MPKI, it is hard to determine efficiently.

The vast majority of the literature is concerned with general-purpose computing, in which the set of tasks changes dynamically and is not known in advance. In this scenario, the characterization of the tasks needs to be performed at

runtime. Qureshi et al. [18] propose dynamic set sampling as an efficient and fairly accurate mechanism to estimate the MPKI. The idea is to focus on a small subset of all cache sets and to extrapolate from the observations in this subset to the entire cache. On this small subset, the cache is augmented with additional tags, which are used to dynamically evaluate the effect of providing more cache ways to a particular task. With these additional tags, the hardware is able to count the potential number of cache hits due to each additional cache way.

In a hard real-time context, the task set is known in advance, and a reasonable characterization would be bounds on the WCET of each task in terms of its partition size.

Given the characterization of each of the tasks, dynamic approaches then predict from the past behavior of the tasks, which choice of partitioning is likely to maximize the chosen performance metric in the future. Depending on the partitioning scheme, a reconfiguration overhead needs to be taken into account. This may be particularly severe in case of software-based partitioning. Zhang et al. [15] discuss tradeoffs in a page coloring approach.

Scheduling Approaches. In addition to choosing good partition sizes for a given workload, system performance may be improved by optimizing the schedule. This has been considered for general-purpose tasks and his particularly relevant in the hard real-time case, as all the necessary information is present at design time.

Zhao et al. [22] present an approach to *dynamic scheduling* that is based on their *CacheScouts* monitoring architecture. This architecture provides hardware performance counters for shared caches that can detect how much cache space individual tasks use, and how much sharing and contention there is between individual tasks. The authors describe three *dynamic* scheduling heuristics that utilize this architecture:

- Schedule a waiting task that has significant sharing with other already running tasks.
- Schedule a task that still has its working set left in the cache.
- Co-schedule tasks with a small and a large working set.

An approach to *static scheduling* in a hard real-time context is presented by Guan et al. [23]. They extend the classical real-time scheduling problem by associating with each task a required cache partition size. They propose an associated scheduling algorithm, *Cache-Aware Non-preemptive Fixed Priority Scheduling*, FP_{CA}. FP_{CA} schedules a job J_i, if

- J_i is the job of highest priority among all waiting jobs,
- there is at least one idle core, and
- enough cache partitions to execute J_i are available.

They propose a linear programming formulation that determines whether a given task set is schedulable under FP_{CA}. For higher efficiency, they also introduce a more efficient heuristic schedulability test that may reject schedulable task sets.

3.2 Summary of the State of the Art and Open Problems

Much work has been done in general-purpose computing to optimize average-case performance by cleverly partitioning shared caches. Much less has been done in the context of hard real-time systems. However, in the case of cache partitioning, hard real-time systems may in fact present a simpler problem: A major remaining challenge is that no real-time scheduling policy taking into account cache space demands is established. The work of Guan et al. [23] is a step in this direction. However, it takes the required cache partition size of each task as an input. Choosing partition sizes to optimize schedulability is an open problem.

In this survey, we have focussed on shared caches. Other storage resources that are increasingly shared in embedded systems are DRAMs and Flash memory. Recently, real-time memory controllers for SDRAM have been proposed that provide bandwidth and latency guarantees to their clients [24,25,26]. These controllers are based on a combination of predictable arbitration mechanisms, such as TDMA, and DRAM-specific access patterns that eliminate the dependence of latencies on the execution history.

4 Assessing the Impact of Resource Sharing on Performance by Measurements

As described earlier in Section 1.1, one possibility to estimate the timing behavior of tasks is by measurements. In this section, we discuss existing measurement-based approaches that aim at quantifying the slowdown a task experiences when different tasks are executed in parallel.

In a single-core setting, a measurement-based estimate is obtained by multiplying the longest observed execution time by a safety margin. However, it is not possible to directly extend such measurement-based timing analyses from the single-core to the multi-core setting.

First, consider timing analysis before tasks have been mapped to cores and scheduled. Given a set of k tasks executing on a system with n cores ($k > n$), there exist $\binom{k}{n}$ possible workloads that could be executed in parallel. Hence, without additional knowledge about the mapping and the schedule, it would be necessary to measure the slowdown of a task in each possible workload in order to determine its worst-case timing. This is very expensive by means of analysis time, especially as all measurements have to be repeated in case the task set changes. Therefore, it is desirable to determine workload-independent estimates of the slowdown. In case of timing analysis after the mapping process, this is not a problem.

Second, the choice of a proper safety margin becomes harder due to the increased variance in execution times. The safety margin is needed to compensate for the potential difference between the highest measured execution time and the actual WCET. With increasing variance, the safety margin must thus be increased as well. Thus the application of the safety margin might lead to a strong

overestimation of the WCET in case the highest measured execution time is close to the actual WCET.

To obtain a workload-independent estimate of the slowdown, so-called resource-stressing benchmarks are employed as co-runners during the measurements. A resource-stressing benchmark is a synthetic program that tries to maximize the load on a resource or a set of resources. The interference a resource-stressing benchmark causes on a certain resource is meant to be an upper bound to the interference any real co-runner could cause. Therefore the slowdown a program experiences due to interferences on a certain resource when co-running with a resource-stressing benchmark bounds the slowdown that could occur with respect to this resource in any real workload.

Typically, resource-stressing benchmarks are independent of the application under consideration, but specific to an architecture. They are independent of the application so they can be reused for the analysis of different programs. The architecture under consideration must be taken into account for properly constructing e.g. a benchmark that maximally stresses the bandwidth to main memory: it must be guaranteed that none of the loads performed by the benchmark can be served by any of the caches. Thus the offset between the addresses has to be chosen in such a way that no cache line is accessed twice. For the proper choice of this parameter, architectural information about the width of a cache line is needed.

As an additional aspect, the obtained slowdowns can be used to estimate the timing predictability of an architecture. A significant slowdown implies a possibly high variance in the execution times of a task and thus disallows accurate timing analysis. Hence, it is also a measure for the suitability of a multi-core architecture for time-critical embedded systems.

In the context of hard real-time embedded systems, measurement-based approaches are inappropriate because they cannot derive safe, analytical guarantees. This is because it cannot be guaranteed that the actual WCET is encountered during the measurements. Even the highest measured execution time together with the safety margin is not necessarily an upper bound to the WCET. However, no static analysis that soundly accounts for all interferences in a multi-core architecture, has been proposed so far.

In Section 4.1, we present the existing approaches in more detail. In Section 4.2, we discuss the shortcomings of the measurement-based approaches and pose open questions.

4.1 Resource-Stressing Benchmarks

Radojković et al. [27], Nowotsch et al. [28] and Fernandez et al. [29] all employ resource-stressing benchmarks in order to quantify the slowdown of tasks in a multi-core architecture. In all three approaches, the respective benchmarks are systematically constructed based on the characteristics of the resource to be stressed. A benchmark that aims at stressing, e.g. parts of the memory hierarchy should almost only involve memory operations and avoid local computations.

The approaches differ in the resources and architectures under consideration as well as in the respective evaluation techniques.

Radojković et al. [27] propose benchmarks that stress a variety of possibly shared resources, including functional units, the memory hierarchy, especially the caches at different levels and the bandwidth to the main memory. Experiments are carried out on three architectures with different shared resources in order to show the varying timing predictability, depending on the degree of resource sharing. One architecture offers hyperthreading, i.e. all resources including the pipeline are shared. For the second architecture, only the bandwidth to the main memory is shared between two cores whereas for the third architecture, the L2 cache is shared as well. In order to show the variance of the possible slowdown, measurements are taken for three different scenarios. In the first scenario, only resource-stressing benchmarks are executed concurrently to estimate the worst possible slowdown independently of the application. In the other cases, the application is either executed with another co-running application or with different co-running resource-stressing benchmarks.

Unsurprisingly, the more resources are shared, the larger is the possible impact of resource sharing on execution times. This makes techniques like hyperthreading impractical for systems with hard real-time constraints. The measurements for the different scenarios reveal that the slowdown due to co-running resource-stressing benchmarks drastically exceeds the slowdown measured in workloads only consisting of real applications. This implies that the workload-independent slowdown determined with co-running resource-stressing benchmarks yields a very imprecise upper bound to the slowdown in any real workload. Therefore, the analysis result might be not very useful in practice.

Nowotsch et al. [28] present benchmarks that stress the memory hierarchy, i.e. the bus to main memory and the caches. The slowdown is measured for workloads exclusively consisting of resource-stressing benchmarks. Several scenarios are considered, testing the influence of different memory configurations (static RAM vs. dynamic RAM) and cache coherency settings on the variance of the observed slowdowns. The overhead due to static coherency (i.e. only checks whether memory blocks in the local caches are coherent) is determined by enabling the coherency flag during measurements although there are no coherent accesses. The overhead due to dynamic coherency (i.e. not only checks, but also explicit memory operations enforcing coherency) is assessed by enabling the coherency flag during measurements in a setting where coherent accesses occur. Measurements are carried out on the Freescale P4080 multi-core processor which is used in avionics.

The outcomes show that the time for concurrent accesses to DDR-SDRAM memory scales very badly with the number of concurrent cores, in contrast to SRAM. Concerning the different cache coherency settings, the results show that even without coherent accesses, static coherency induces an overhead in execution time that should be considered in timing analysis. The impact of dynamic coherency strongly depends on the type of memory operation (read or write). In case of read with concurrent read, dynamic coherency does not cause any

slowdown compared to static coherency. For write operations, the execution is slowed down significantly in case of dynamic coherency, regardless of the concurrent operation. This can be explained by the fact that after a write operation, coherency actions are required to keep the memory hierarchy consistent.

The benchmarks used by Fernandez et al. [29] focus on the memory hierarchy of the system, including the caches at different levels as well as the memory controller. In order to evaluate the influence of the underlying operating system on the slowdown due to shared resources, the measurements are performed once on Linux and once on the real-time operating system RTEMS. The overhead in execution time is determined for two types of workloads: task sets exclusively composed of resource-stressing benchmarks and task sets with one application and one co-running resource-stressing benchmark.

The outcomes of the experiments with benchmarks that exclusively stress the private L1 cache through store operations show that a considerable slowdown is produced. This is due to the fact that the L1 cache is write-through, thus store operations lead to bus traffic, L2 cache interference and memory controller accesses. The number of store instructions within an application is identified to be the dominant metric for slowdowns in this architecture. Employing write-back caches reduces this overhead but introduces new overhead arising from cache coherency protocols, as it has been described in [28]. The results produced for the different operating systems show that they have a non-negligible influence as well. For example, whereas the estimated miss rate is 11% for a certain application in a Linux environment, the same application shows a miss-rate of near zero on the RTEMS operating system. Similar to the outcomes of the other papers, the application-independent slowdown measured in workloads with only resource-stressing benchmarks exceeds the application-dependent one.

4.2 Summary of the State of the Art and Open Problems

The state of the art approaches provide measurement-based estimates on the slowdown of a task due to interferences on shared resources caused by co-running tasks. These estimates are then used to roughly classify components of a multi-core architecture with respect to timing predictability. There are two main concerns about the soundness of the measurement-based approaches employing resource-stressing benchmarks.

First, there is neither a proof nor a formal argument why a specific benchmark puts maximal load on a resource. For the storage resources in particular, it is very difficult to argue why their state is affected in the worst possible way. In case of several possible co-runners, these arguments become even more difficult because of the combined interferences.

Second, a resource-stressing benchmark is typically application-independent. An application with memory-intensive as well as computation-intensive parts is neither slowed down maximally by a cache-stressing nor by a functional-unit-stressing co-runner. To account for this behavior, either a sound combination of slowdowns obtained by different benchmarks or an application-specific benchmark is needed. Obtaining a sound combination of slowdowns, which we call

compositionality issue, is hard because resources are not independent of each other. For example, a program that stresses the memory bus does also necessarily stress the last-level-cache. But, a benchmark that stresses several resources simultaneously cannot stress one particular resource in the worst possible way at the same time. We call the application-specific benchmark that causes the maximal slowdown *worst companion*. In general, the construction of this worst companion might be infeasible due to limited control over the hardware itself. Nevertheless, application-specific benchmarks might produce worse slowdowns than application-independent benchmarks which disproves the soundness of these measurement-based techniques.

Beyond the open questions concerning the soundness of the approaches, the usefulness of the obtained slowdowns is questionable. While a resource-stressing benchmark can cause slowdown factors of up to 20, in reality, applications are co-located with other applications that typically cause much less interferences on shared resources. Radojković et al. [27] demonstrate that the slowdown an application experiences with co-running resource-stressing benchmarks is significantly higher compared to other co-running applications.

An orthogonal research problem is the construction of supporting benchmarks that e.g. use cache prefetching to speed up the application. Analogously to the worst companion, one could aim at the construction of a *best companion*.

5 Conclusions

This survey has given an overview of the impact of the interference on shared resources on performance and performance estimation. Goals of the described approaches were performance estimation or performance improvement. Mainly, two types of resources were considered, bandwidth resources, e.g. buses, and storage resources, e.g. caches.

Performance estimation methods for multi-core systems with shared buses attempt to derive bounds on the overall access delays under different arbitration protocols. All use cumulative abstractions such as the number of bus accesses during bounded-length phases in tasks in a simplified task model. Different arbitration protocols have different worst-case scenarios in which bounds on the access delays are computed.

Static analysis of the cache behavior of shared caches is highly complex due to the large number of potential interleavings to be considered. Cache partitioning among the different tasks on different cores is used instead. Methods known from the single-core case can then be used. Performance of statically shared caches will suffer for systems with dynamically varying demands on memory. Dynamic cache partitioning is the answer. Existing approaches consider shared caches in isolation and ignore the effect cache reloads and write backs have on the necessarily shared bus.

Several approaches attempt to construct resource-stressing benchmarks. Such a benchmark for a specific resource is meant to cause the maximal slowdown on a co-running application. In an interval of bounded length any resource can only be stressed to a degree that is a function of the interval length. The presented

ways to design resource-stressing benchmarks do this in an intuitive, but ad-hoc way. They are independent of any particular application. It seems clear that a larger slowdown of a particular application can be achieved by an application-specific *worst companion*, which better exploits the available time to exercise stress on shared resources. Thus the existing resource-stressing benchmarks do not exhibit upper bounds on the interference. On the other hand, they are overly pessimistic: a co-running application will seldom exercise such a large stress on a shared resource.

References

1. Ungerer, T., Cazorla, F., Sainrat, P., Bernat, G., Petrov, Z., Rochange, C., Quinones, E., Gerdes, M., Paolieri, M., Wolf, J., Casse, H., Uhrig, S., Guliashvili, I., Houston, M., Kluge, F., Metzlaff, S., Mische, J.: Merasa: Multicore execution of hard real-time applications supporting analyzability. IEEE Micro 30, 66–75 (2010)
2. Wilhelm, R., Grund, D., Reineke, J., Schlickling, M., Pister, M., Ferdinand, C.: Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 28(7), 966–978 (2009)
3. FlexRay Consortium: FlexRay, `http://www.flexray.com/`
4. BMW: BMW technology guide: Flex Ray, `http://www.bmw.com/com/en/insights/technology/technology_guide/articles/flex_ray.html`
5. Robert Bosch GmbH: FlexRay communication controller IP, `http://www.bosch-semiconductors.de/en/ipmodules/flexray/flexray.asp`
6. Schranzhofer, A., Chen, J.J., Thiele, L.: Timing analysis for TDMA arbitration in resource sharing systems. In: Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2010, pp. 215–224. IEEE Computer Society, Washington, DC (2010)
7. Pellizzoni, R., Caccamo, M.: Impact of peripheral-processor interference on WCET analysis of real-time embedded systems. IEEE Transactions on Computers 59, 400–415 (2010)
8. Pellizzoni, R., Schranzhofer, A., Chen, J.J., Caccamo, M., Thiele, L.: Worst case delay analysis for memory interference in multicore systems. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2010, 3001 Leuven, Belgium, pp. 741–746. European Design and Automation Association (2010)
9. Schranzhofer, A., Pellizzoni, R., Chen, J.J., Thiele, L., Caccamo, M.: Timing analysis for resource access interference on adaptive resource arbiters. In: Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, pp. 213–222. IEEE Computer Society, Washington, DC (2011)
10. Rosen, J., Andrei, A., Eles, P., Peng, Z.: Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In: Proceedings of the 28th IEEE International Real-Time Systems Symposium, RTSS 2007, pp. 49–60. IEEE Computer Society, Washington, DC (2007)
11. Boniol, F., Cassé, H., Noulard, E., Pagetti, C.: Deterministic execution model on COTS hardware. In: Herkersdorf, A., Römer, K., Brinkschulte, U. (eds.) ARCS 2012. LNCS, vol. 7179, pp. 98–110. Springer, Heidelberg (2012)

12. Cruz, R.L.: A calculus for network delay. IEEE Transactions on Information Theory 37(1), 114–141 (1991)
13. Wandeler, E., Thiele, L., Verhoef, M., Lieverse, P.: System architecture evaluation using modular performance analysis: a case study. Int. J. Softw. Tools Technol. Transf. 8(6), 649–667 (2006)
14. Reineke, J., Grund, D., Berg, C., Wilhelm, R.: Timing predictability of cache replacement policies. Real-Time Systems 37(2), 99–122 (2007)
15. Zhang, X., Dwarkadas, S., Shen, K.: Towards practical page coloring-based multicore cache management. In: Proceedings of the 4th ACM European conference on Computer systems, EuroSys 2009, pp. 89–102. ACM, New York (2009)
16. Suhendra, V., Mitra, T.: Exploring locking & partitioning for predictable shared caches on multi-cores. In: Proceedings of the 45th Annual Design Automation Conference, DAC 2008, pp. 300–303. ACM, New York (2008)
17. Nesbit, K.J., Laudon, J., Smith, J.E.: Virtual private caches. SIGARCH Comput. Archit. News 35(2), 57–68 (2007)
18. Qureshi, M.K., Patt, Y.N.: Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In: IEEE/ACM International Symposium on Microarchitecture, MICRO 2006, pp. 423–432. IEEE Computer Society (2006)
19. Xie, Y., Loh, G.H.: PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In: Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA 2009, pp. 174–183. ACM, New York (2009)
20. Taylor, G., Davies, P., Farmwald, M.: The TLB slice – a low-cost high-speed address translation mechanism. SIGARCH Comput. Archit. News 18(3a), 355–363 (1990)
21. Tam, D., Azimi, R., Soares, L., Stumm, M.: Managing shared L2 caches on multicore systems in software. In: Workshop on the Interaction between Operating Systems and Computer Architecture (2007)
22. Zhao, L., Iyer, R., Illikkal, R., Moses, J., Makineni, S., Newell, D.: CacheScouts: Fine-grain monitoring of shared caches in CMP platforms. In: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, PACT 2007, pp. 339–352. IEEE Computer Society, Washington, DC (2007)
23. Guan, N., Stigge, M., Yi, W., Yu, G.: Cache-aware scheduling and analysis for multicores. In: Proceedings of the Seventh ACM International Conference on Embedded Software, EMSOFT 2009, pp. 245–254. ACM, New York (2009)
24. Akesson, B., Goossens, K., Ringhofer, M.: Predator: a predictable SDRAM memory controller. In: CODES+ISSS, pp. 251–256. ACM (2007)
25. Paolieri, M., Quiñones, E., Cazorla, F., Valero, M.: An analyzable memory controller for hard real-time CMPs. IEEE Embedded Systems Letters 1(4), 86–90 (2010)
26. Reineke, J., Liu, I., Patel, H.D., Kim, S., Lee, E.A.: PRET DRAM controller: bank privatization for predictability and temporal isolation. In: Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2011, pp. 99–108. ACM, New York (2011)
27. Radojković, P., Girbal, S., Grasset, A., Quiñones, E., Yehia, S., Cazorla, F.J.: On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. ACM Trans. Archit. Code Optim. (2012)
28. Nowotsch, J., Paulitsch, M.: Leveraging multi-core computing architectures in avionics. In: EDCC (2012)
29. Fernandez, M., Gioiosa, R., Quiñones, E., Fossati, L., Zulianello, M., Cazorla, F.J.: Assessing the suitability of the NGMP multi-core processor in the space domain. In: EMSOFT (2012)

Concurrency Meets Probability: Theory and Practice (Abstract)

Joost-Pieter Katoen

Software Modelling and Verification, RWTH Aachen University, Germany
Formal Methods and Tools, University of Twente, The Netherlands

Treating random phenomena in concurrency theory has a long tradition. Petri nets [18,10] and process algebras [14] have been extended with probabilities. The same applies to behavioural semantics such as strong and weak (bi)simulation [1], and testing pre-orders [5]. Beautiful connections between probabilistic bisimulation [16] and Markov chain lumping [15] have been found. A plethora of probabilistic concurrency models has emerged [19]. Over the years, the focus shifted from covering discrete to treating continuous stochastic phenomena [12,13].

We argue that both aspects can be elegantly combined with non-determinism, yielding the *Markov automata* model [8]. This model has nice theoretical characteristics. It is closed under parallel composition and hiding. Conservative extensions of (bi)simulation are congruences [8,4]. It has a simple process algebraic counterpart [20]. On-the-fly partial-order reduction yields substantial state-space reductions [21]. Their quantitative analysis largely depends on (efficient) linear programming and scales well [11].

More importantly though: Markov automata serve an important *practical need*. They are the obvious choice for providing semantics to the Architecture Analysis & Design Language (AADL [9]), an industry standard for the automotive and aerospace domain. As experienced in several ESA projects, this holds in particular for the AADL annex dealing with error models [3]. They provide a compositional semantics to dynamic fault trees [6], a key model for reliability engineering [2]. Finally, they give a natural semantics to *every* generalised stochastic Petri net (GSPN [17]), a prominent model in performance analysis. This conservatively extends the existing GSPN semantics that is restricted to "well-defined" nets, i.e., nets without non-determinism [7]. Powerful software tools support this and incorporate efficient analysis and minimisation algorithms [11].

This substantiates our take-home message: *Markov automata bridge the gap between an elegant theory and practical engineering needs.*

Acknowledgement. This work is funded by the EU FP7-projects MoVeS, SENSATION and MEALS, the DFG-NWO bilateral project ROCKS, the NWO project SYRUP, the ESA project HASDEL, and the STW project ArRangeer.

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 44–45, 2013.

References

1. Baier, C., Katoen, J.-P., Hermanns, H., Wolf, V.: Comparative branching-time semantics for Markov chains. Inf. Comput. 200(2), 149–214 (2005)
2. Boudali, H., Crouzen, P., Stoelinga, M.: A rigorous, compositional, and extensible framework for dynamic fault tree analysis. IEEE Trans. Dependable Sec. Comput. 7(2), 128–143 (2010)
3. Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V.Y., Noll, T., Roveri, M.: Safety, dependability and performance analysis of extended AADL models. The Computer Journal 54(5), 754–775 (2011)
4. Deng, Y., Hennessy, M.: On the semantics of Markov automata. Inf. Comput. 222, 139–168 (2013)
5. Deng, Y., van Glabbeek, R.J., Hennessy, M., Morgan, C.: Characterising testing preorders for finite probabilistic processes. Logical Methods in Computer Science 4(4) (2008)
6. Dugan, J., Bavuso, S.: Dynamic fault-tree models for fault-tolerant computer systems. IEEE Tr. on Reliability 41(3), 363–377 (1992)
7. Eisentraut, C., Hermanns, H., Katoen, J.-P., Zhang, L.: A semantics for every GSPN. In: Colom, J.-M., Desel, J. (eds.) PETRI NETS 2013. LNCS, vol. 7927, pp. 90–109. Springer, Heidelberg (2013)
8. Eisentraut, C., Hermanns, H., Zhang, L.: On probabilistic automata in continuous time. In: LICS, pp. 342–351. IEEE Computer Society (2010)
9. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL - An Introduction to the SAE Architecture Analysis and Design Language. SEI Series in Software Engineering. Addison-Wesley (2012)
10. Florin, G., Natkin, S.: Les reseaux de Petri stochastiques. Technique et Science Informatiques 4(1), 143–160 (1985)
11. Guck, D., Hatefi, H., Hermanns, H., Katoen, J.-P., Timmer, M.: Modelling, reduction and analysis of Markov automata. In: Joshi, K., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) QEST 2013. LNCS, vol. 8054, pp. 55–71. Springer, Heidelberg (2013)
12. Hermanns, H., Herzog, U., Katoen, J.-P.: Process algebra for performance evaluation. Theor. Comput. Sci. 274(1-2), 43–87 (2002)
13. Hillston, J.: Process algebras for quantitative analysis. In: LICS, pp. 239–248. IEEE Computer Society (2005)
14. Jonsson, B., Yi, W., Larsen, K.G.: Probabilistic extensions of process algebras. In: Bergstra, J., Ponse, A., Smolka, S. (eds.) Handbook of Process Algebra, ch. 11, pp. 685–711 (2001)
15. Kemeny, J., Snell, J.: Finite Markov Chains. D. Van Nostrand (1960)
16. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Inf. Comput. 94(1), 1–28 (1991)
17. Marsan, M.A., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets. John Wiley & Sons (1995)
18. Molloy, M.K.: Performance analysis using stochastic Petri nets. IEEE Trans. Computers 31(9), 913–917 (1982)
19. Sokolova, A., de Vink, E.P.: Probabilistic automata: System types, parallel composition and comparison. In: Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P., Siegle, M. (eds.) Validation of Stochastic Systems. LNCS, vol. 2925, pp. 1–43. Springer, Heidelberg (2004)
20. Timmer, M., Katoen, J.-P., van de Pol, J., Stoelinga, M.: Efficient modelling and generation of Markov automata. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 364–379. Springer, Heidelberg (2012)
21. Timmer, M., van de Pol, J., Stoelinga, M.I.A.: Confluence Reduction for Markov Automata. In: Braberman, V., Fribourg, L. (eds.) FORMATS 2013. LNCS, vol. 8053, pp. 243–257. Springer, Heidelberg (2013)

Modular Semantics for Transition System Specifications with Negative Premises

Martin Churchill[1], Peter D. Mosses[1], and Mohammad Reza Mousavi[2]

[1] Department of Computer Science, Swansea University
{m.d.churchill,p.d.mosses}@swansea.ac.uk
[2] Halmstad University and Eindhoven University of Technology
m.r.mousavi@hh.se

Abstract. Transition rules with negative premises are needed in the structural operational semantics of programming and specification constructs such as priority and interrupt, as well as in timed extensions of specification languages. The well-known proof-theoretic semantics for transition system specifications involving such rules is based on well-supported proofs for closed transitions. Dealing with open formulae by considering all closed instances is inherently non-modular – proofs are not necessarily preserved by disjoint extensions of the transition system specification.

Here, we conservatively extend the notion of well-supported proof to open transition rules. We prove that the resulting semantics is modular, consistent, and closed under instantiation. Our results provide the foundations for modular notions of bisimulation such that equivalence can be proved with reference only to the relevant rules, without appealing to all existing closed instantiations of terms.

1 Introduction

The main goal of this paper is to provide *modular* proof theory for structural operational semantics when transition rules with negative premises are allowed. The main technical contributions are a notion of well-supported proof for open transition rules, together with theorems that establish various essential properties of this notion. This is part of our larger research effort in defining a modular semantic framework, including machinery such as bisimulation proof techniques [15], and rule formats for the operational semantics of programming and specification languages which ensure that bisimilarity is a congruence [7].

When Plotkin introduced structural operational semantics (SOS) in his seminal Aarhus lecture notes in 1981 [17], he used only positive transition rules: the possibility of a transition for a programming construct depended on the possibility of transitions for its sub-constructs – never on their impossibility. In that context, the transition relation defined by a set of SOS rules is always well-defined, and the proof theory of transitions is quite straightforward (except regarding modularity of bisimilarity; see [15]). Positive transition rules are adequate for specifying the SOS of many programming and specification language constructs.

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 46–60, 2013.

Nevertheless, negative premises have been found useful in SOS. For example, when termination can be conflated with deadlock (as in some process algebras) the following transition rules specify sequential execution of the construct $(x; y)$:

$$\frac{x \xrightarrow{l} x'}{(x; y) \xrightarrow{l} (x'; y)} \qquad \frac{x \overset{l'}{\not\rightarrow} \quad y \xrightarrow{l} y'}{(x; y) \xrightarrow{l} y'}$$

This avoids the need to introduce distinguished terminal states, or a termination predicate.

More significantly, it has been shown [3] that transition rules with negative premises are actually *necessary* for the SOS of some programming and specification constructs, such as priority operators: SOS is strictly more expressive when negative premises are allowed. Related examples where negative premises are needed include interrupts and timed extensions of specification languages.

The model- and proof-theoretic semantics for SOS specifications involving negative premises is considerably less obvious than in the positive case; see [13,2,12] for detailed discussions and comparison of alternative definitions. A widely accepted definition is based on *well-supported proofs* for transition formulae $p \xrightarrow{l} q$ where p (and q) are *closed* terms [12,5]. Well-supported-proofs for open formulae has remained an open problem since 1995 [11] (and the task was characterised as 'somewhat problematic' by Van Glabbeek [12]). In the negative setting, the usual *closed-instance semantics* for open formulae would allow $r \overset{l}{\not\rightarrow}$ to be inferred whenever it is impossible to infer $r \xrightarrow{l} u$ for any u (corresponding to 'negation as failure' in logic programming [8]). But this is inherently *non-modular*: proofs are not generally preserved when the transition system specification is extended with new constructs and with rules defining the transitions of the new constructs. The non-modularity stems from defining the notion of well-supported proof with respect to the set of all closed terms in a language: extensions of the specified language increase that set.

In this paper, we conservatively extend the notion of well-supported proof to *open transition rules*, in contrast to closed-instance semantics. We prove that the resulting semantics is modular, consistent, and closed under instantiation.

The conservativeness of our semantics requires a mild condition on the format of transition rules: *source-dependency*, which (informally stated) ensures that each variable in a rule can be traced back to variables that occur in the source of the conclusion (via transitions in the premises of the rule). Source-dependency was also required to show that disjoint extensions are operationally conservative with respect to closed transition formulae in [10]. Our other results (including modularity) apply to arbitrary specifications.

The work here provides foundations for modular notions of bisimulation for systems with negative premises, whereby equivalence between two terms can be proved with reference only to the rules that define transitions for the constructs occurring in those terms (independently of the presence or absence of other constructs and their defining rules). Modular bisimulation proofs correspond

closely to conventional proofs which appeal to the fact that 'no further rules need to be considered'.

The rest of this paper is organised as follows. In Section 2, we recall some standard notions. In Section 3, we generalise the notion of well-supported proof to open transition rules. We show that our notion of well-supported proof is consistent (i.e., does not lead to proofs of denying formulae) and closed under instantiation of formulae and transition rules. In Section 4, we study the issue of modularity. First, we show that the usual notion of closed instance semantics is not modular, in general. Second, we show that our approach to assigning semantics to open formulae is indeed modular. In Section 5, we show that our notion of semantics is a conservative extension of the existing notion for closed terms (i.e., it leads to the same set of provable transitions for closed terms), and that disjoint extensions are conservative. This requires the mild condition of source-dependency. We conclude the paper and present some direction for future work in Section 6.

2 Preliminaries

We begin by recalling some standard definitions regarding SOS specifications from the literature (see [2,16] for further details).

Definition 1 (Signatures, Terms and Substitutions). *We assume a countable set V of* variables*. A* signature Σ *is a set of function symbols with fixed arities; the arity of f is a non-negative integer denoted by $ar(f)$. The set of* terms *on signature Σ, denoted by $\mathbb{T}(\Sigma)$ and ranged over by $s, t, s_0, t_0, \ldots$, is defined inductively as follows: variables and function symbols of arity zero (also called constants) are terms; given a list of terms, their composition using a function symbol (while respecting the arity of the function symbol) is a term. Terms are also called* open terms*; the set of variables in t is denoted by $\mathsf{vars}(t)$.* Closed terms *on signature Σ, denoted by $\mathbb{C}(\Sigma)$ and ranged over by $p, q, \ldots$, are those terms in $\mathbb{T}(\Sigma)$ that do not contain any variable. A* substitution $\sigma : V \to \mathbb{T}(\Sigma)$ *is a function from variables to terms; it is* closing *if it maps variables to closed terms. These are lifted to functions on terms in the usual manner. We write ι for the identity substitution, and if σ is a substitution, write $\sigma[x \mapsto s]$ for the substitution that sends x to s and other variables y to $\sigma(y)$.*

Transition System Specifications (TSSs), introduced in [14,6], are formalisations of SOS specifications. Here, we consider TSSs where positive formulae are restricted to labelled transitions $s \xrightarrow{l} t$; extension to allow multiple transition relations and other predicates would be straightforward.

Definition 2 (Transition System Specification). *A* transition system specification T *is a tuple (Σ, L, D) where Σ is a signature, L is a set of labels (with typical members $a, b, a_0, \ldots$) and D is a set of deduction rules. For all $l \in L$, and $t, t' \in \mathbb{T}(\Sigma)$ we define that $t \xrightarrow{l} t'$ is a* positive formula *and $t \xnrightarrow{l}$ is a* negative formula*; t is the* source *of both formulae and t' is the* target *of the former. A*

formula *is either a positive or a negative formula. For each* t, t'*, the formula* $t \overset{a}{\not\to}$ denies $t \overset{a}{\longrightarrow} t'$ *and vice versa. A formula is* closed *when all terms appearing in it are closed. A* deduction rule $d \in D$ *is a pair* (H, ϕ)*, where* H *is a set of formulae and* ϕ *is a positive formula;* ϕ *is called the* conclusion *and the formulae in* H *are called* premises*. A deduction rule is* f-defining *when the source of its conclusion is of the form* $f(s_1, \ldots, s_n)$*. A deduction rule is an* axiom *when it has no premises, and* closed *when all formulae appearing in it are closed.*

We sometimes refer to a TSS by its set of deduction rules. A deduction rule (H, ϕ) is also written as $\frac{H}{\phi}$; in the latter syntax, if H is empty then it may be omitted.

We next recall the standard notion of proofs in TSSs with negative premises [12], to be generalised to open terms in the rest of this paper.

Definition 3 (Derivation). *A* derivation π *for* $\frac{H}{\phi}$ *in a TSS* T *is a well-founded upwardly branching tree with nodes labelled by formulae of* T *and of which*

- *the root is labelled by* ϕ*;*
- *if a node is labelled by* ψ *and the nodes immediately above it form the set* K *then:*
  - $\psi \in H$ *and* $K = \emptyset$*, or*
  - ψ *is a positive formula and* $\frac{K}{\psi}$ *is a substitution instance of a deduction rule in* T*.*

A derivation is closed *if all nodes are labelled with a closed formula. A formula* occurs in *a derivation if it labels a node in that derivation. We lift the application of substitutions to derivations in the usual way.*

Definition 4 (Provable Rule). *A closed deduction rule* $\frac{H}{\phi}$ *is a* provable rule *if it has a closed derivation* π*.*

Definition 5 (Ground Well-Supported Proof). *If* ϕ *is a closed formula, a* ground well-supported proof for ϕ *in a TSS* T *is a well-founded upwardly branching tree with nodes labelled by closed formulae and of which*

- *the root is labelled by* ϕ*;*
- *if a node is labelled by* ψ *and the nodes immediately above it form the set* K *then:*
  - ψ *is a positive formula and* $\frac{K}{\psi}$ *is an instance of a deduction rule in* T*, or*
  - ψ *is a negative formula and, for each set* N *of closed negative formulae and each* ψ' *denying* ψ *such that* $\frac{N}{\psi'}$ *is a provable rule, there is a formula in* N *denying a formula in* K*.*

The above definition corresponds to Definition 12 in [12].

3 Well-Supported Proofs

In this section, we generalise the notion of well-supported proof from closed formulae to *open rules*.

3.1 Provable Ruloids and Well-Supported Proofs

In order to build up a proof tree for $\frac{H}{\phi}$, one must provide justification for the to-be-proven formulae, until reaching a premise in H. For the positive formulae in such a proof tree, we require them to be justified using the deduction rules in the TSS. For the negative formulae, we consider *provable ruloids*: a generalisation of the notion of provable rule from closed to open rules.

Definition 6 (Provable Ruloid). *A context is a set* $\{x_i \xrightarrow{l_i} s_i, t_j \overset{l_j}{\nrightarrow} \mid i \in I, j \in J\}$ *of formulae (for possibly empty sets of indices* I *and* J*). A deduction rule* $\frac{H}{\phi}$ *is a* provable ruloid *if* H *is a context and* $\frac{H}{\phi}$ *has a derivation* π*. We say that* π witnesses *the provable ruloid* $\frac{H}{\phi}$*. A derivation* π *is a* provable ruloid derivation *if it witnesses some provable ruloid, i.e., each leaf with a positive formula has a variable as its source.*

The arbitrary negative formulae appearing in contexts and leaf positions of provable ruloid derivations correspond to the set N in Definition 5.

We next generalise the definition of well-supported proof to the open setting, in the presence of a set of hypotheses asserting the possibility or impossibility of transitions from variables (so-called GSOS [4] contexts). We may discharge proof obligations for a negative formula by appealing to an appropriate hypothesis or by denying its possible proofs. In the open setting, such possible proofs may conclude substitutive instances of the formula in question.

Definition 7 (Well-Supported Proof). *A context* H *is called a GSOS context if the source of each formula in* H *(in particular, the negative ones) is a variable. For a GSOS context* H *and formula* ϕ*, a* well-supported proof *for* $\frac{H}{\phi}$ *in a TSS* T *is a well-founded upwardly branching tree with nodes labelled by formulae and of which*

- *the root is labelled by* ϕ*;*
- *if a node is labelled by* ψ *and the nodes immediately above it form the set* K *then:*
  - $\psi \in H$ *and* $K = \emptyset$*, or*
  - ψ *is a positive formula and* $\frac{K}{\psi}$ *is an instance of a deduction rule in* T*, or*
  - ψ *is a negative formula and for each substitution* σ*,* ψ' *denying* $\sigma(\psi)$ *and provable ruloid derivation* π *concluding* ψ'*, there exists* $\kappa \in K$ *and* κ' *occurring in* π *such that* κ' *denies* $\sigma(\kappa)$*.*

If $\frac{H}{\phi}$ has a well-supported proof, we write that $\frac{H}{\phi}$ is *(ws-)provable*. A well-supported proof is *closed* if it contains only closed formulae.

Remark 8. *In any TSS, $\overline{x \stackrel{l}{\not\rightarrow}}$ does not have a well-supported proof. For suppose it did, and consider the smallest such proof, with conclusion $x \stackrel{l}{\not\rightarrow}$ and immediate premises K. Then $x \stackrel{l}{\longrightarrow} x$ denies $\iota(x \stackrel{l}{\not\rightarrow})$, and $\frac{x \stackrel{l}{\longrightarrow} x}{x \stackrel{l}{\longrightarrow} x}$ is a provable ruloid, witnessed by a derivation π with a single node $x \stackrel{l}{\longrightarrow} x$. Hence there exists $\kappa \in K$ and κ' denying $\iota(\kappa) = \kappa$ occurring in π. But the only formula κ' occurring in π is $x \stackrel{l}{\longrightarrow} x$ and we must have $\kappa = x \stackrel{l}{\not\rightarrow}$. Hence, there exists another (smaller) proof for $\overline{x \stackrel{l}{\not\rightarrow}}$ in the original proof; this contradicts the assumption that we started from the smallest such proof.*

The above fact is crucial for modularity: the TSS may be extended with new constructs (and rules for them) which violate the general formula $x \stackrel{l}{\not\rightarrow}$, and we wish the old proofs to remain valid as the TSS is extended. The notion of negative proof search used in our notion of well-supported proof does not admit exhaustive case analysis on the possible instantiations of the variables.

Our definition of well-supported proof (Definition 7) differs from the closed notion (Definition 5) in some important respects, as illustrated by the following examples. However, in Section 5 we will show that for closed ϕ in a source-dependent TSS, $\overline{\phi}$ is ws-provable if and only if $\overline{\phi}$ has a ground well-supported proof.

Example 9. *Consider a TSS with unary symbols f, g; constants 0 and 1; label a; and deduction rules $\frac{f(x) \stackrel{a}{\not\rightarrow}}{g(x) \stackrel{a}{\longrightarrow} x}$, $\overline{f(0) \stackrel{a}{\longrightarrow} 0}$. Then:*

- *$\overline{f(1) \stackrel{a}{\not\rightarrow}}$ is provable as there are no provable ruloids concluding $\sigma(f(1) \stackrel{a}{\longrightarrow} y)$. Thus, $\overline{g(1) \stackrel{a}{\longrightarrow} 1}$ is also provable.*
- *Since $f(0) \stackrel{a}{\longrightarrow} 0$ is a provable ruloid derivation, neither $\overline{f(0) \stackrel{a}{\not\rightarrow}}$ nor $\overline{g(0) \stackrel{a}{\longrightarrow} 0}$ are provable.*
- *$\overline{f(x) \stackrel{a}{\not\rightarrow}}$ is not provable, due to the provable ruloid derivation $\overline{f(0) \stackrel{a}{\longrightarrow} 0}$ concluding $\iota[x \mapsto 0](f(x) \stackrel{a}{\longrightarrow} 0)$. Thus, $\overline{g(x) \stackrel{a}{\longrightarrow} x}$ is not provable.*

The above example demonstrates why we must consider counterexamples up to substitution: otherwise, $\overline{f(x) \stackrel{a}{\not\rightarrow}}$ and $\overline{g(x) \stackrel{a}{\longrightarrow} x}$ would indeed be provable, but $\overline{g(0) \stackrel{a}{\longrightarrow} 0}$ unprovable – provability would not be closed under instantiation, which is counter-intuitive.

Example 10. *Consider a TSS with constant 0, unary f, labels a and b, and deduction rule $\frac{x \stackrel{a}{\longrightarrow} 0}{f(x) \stackrel{b}{\longrightarrow} 0}$. Then $\frac{x \stackrel{a}{\not\rightarrow}}{f(x) \stackrel{b}{\not\rightarrow}}$ is provable. Each ϕ that denies $\sigma(f(x) \stackrel{b}{\not\rightarrow})$ is of the form $\sigma(f(x) \stackrel{b}{\longrightarrow} s)$ and the only provable ruloid derivation concluding*

this is $\dfrac{\sigma(x) \xrightarrow{a} 0}{f(\sigma(x)) \xrightarrow{b} 0}$. *But* $\sigma(x \xrightarrow{a} 0)$ *occurs in this derivation, denying* $\sigma(x \overset{a}{\nrightarrow})$, *as required.*

If we extend the TSS with an additional symbol 1 *with* $\overline{1 \xrightarrow{a} 0}$ *then* $\dfrac{x \overset{a}{\nrightarrow}}{f(x) \overset{b}{\nrightarrow}}$ *remains provable. This time, if* $\sigma(x) = 1$, *there is an additional provable ruloid derivation concluding* $\sigma(f(x) \xrightarrow{b} s)$ *to consider:* $\dfrac{\overline{1 \xrightarrow{a} 0}}{f(1) \xrightarrow{b} 0}$. *But* $1 \xrightarrow{a} 0$ *occurs in this provable ruloid, which denies* $\sigma(x \overset{a}{\nrightarrow})$, *as required.*

The above example demonstrates why in Definition 7 we must allow κ' to occur in a non-leaf position of π. Otherwise, the proof of $\dfrac{x \overset{a}{\nrightarrow}}{f(x) \overset{a}{\nrightarrow}}$ would become invalid after extending by an unrelated constant 1, and modularity would fail.

Unlike the closed case, the provable ruloid derivations we consider may have positive leaves whose source is a variable. This is to allow negative information about variables to pass from the well-supported proofs to the provable ruloids. One might consider restricting negative leaves to those whose source is a variable (i.e., to GSOS contexts), but this would lead to an inconsistent notion of proof, as the next example shows.

Example 11. *Consider the TSS with the signature containing constant* 0, *unary function symbol* f, *label* a, *and deduction rule* $\dfrac{x \overset{a}{\nrightarrow}}{f(x) \xrightarrow{a} f(x)}$.

Then $\overline{f^{2n+1}(0) \xrightarrow{a} f^{2n+1}(0)}$ *is provable for each* $n \in \mathbb{N}$, *by a simple induction on* n.

Now, consider the formula $f^3(0) \overset{a}{\nrightarrow}$; *in order to prove it, one needs to find all provable ruloid derivations concluding* $f^3(0) \xrightarrow{a} t$ *(for some term* t*) and deny an occurring formula in each and every derivation. The only provable ruloid derivation with* $f^3(0) \xrightarrow{a} t$ *as its conclusion is* $\dfrac{f^2(0) \overset{a}{\nrightarrow}}{f^3(0) \xrightarrow{a} f^3(0)}$. *Thus, if one only allowed provable ruloid derivations from GSOS contexts,* $\overline{f^3(0) \overset{a}{\nrightarrow}}$ *would be provable as well as* $\overline{f^3(0) \xrightarrow{a} f^3(0)}$, *and consistency would fail.*

In the rest of this paper, we show that Definition 7 supports instantiation closure, consistency, modularity, and that (under the mild but necessary condition of source-dependency) disjoint extensions are conservative.

3.2 Basic Results

We first show that our notion of well-supported proof is consistent: it cannot be the case that both $\overline{\phi}$ and $\overline{\phi'}$ have well-supported proofs for denying ϕ and ϕ'. Since proofs for open formulae occur with respect to GSOS contexts, we generalise this notion of consistency to "consistent" contexts, i.e., contexts that do not

themselves contain a contradiction. In addition, the TSS should satisfy a sanity condition: it should not induce non-trivial deduction rules concluding formulae whose conclusion source is a variable. If it did, this can lead to contradiction when combined with $\underline{\text{GSOS}}$ contexts as proof hypotheses. For example, in a TSS with deduction rule $\frac{}{x \xrightarrow{l} x}$, any assumption of the form $x \overset{l}{\not\to}$ yields inconsistency – both $\frac{x \overset{l}{\not\to}}{x \overset{l}{\not\to}}$ and $\frac{x \overset{l}{\not\to}}{x \xrightarrow{l} x}$ have well-supported proofs. (In such pathological systems, consistency can still be recovered under positive GSOS contexts.) These requirements are captured in the following two definitions.

Definition 12 (Consistent Contexts). *A GSOS context is* consistent *if for each x, l, s, it does not contain both $x \xrightarrow{l} s$ and $x \overset{l}{\not\to}$.*

Definition 13 (Lean TSSs). *A TSS is* lean *if for variables x, $\frac{H}{x \xrightarrow{l} s}$ is only provable when $x \xrightarrow{l} s \in H$.*

Now, we have the ingredients to recast the consistency result in the setting with open terms.

Theorem 14 (Consistency). *Consider a TSS $T = (\Sigma, L, D)$ and consistent GSOS context H. Suppose further that T is lean,* or *H contains only positive formulae. Let ϕ and ϕ' be denying formulae. Then it is not the case that both $\frac{H}{\phi}$ and $\frac{H}{\phi'}$ have well-supported proofs.*

Proof. Assume that both ϕ and ϕ' are provable from H by well-supported proofs π and π' respectively. Assume without loss of generality that ϕ' is a negative formula. We will seek a contradiction, proceeding by induction on the total depth of π and π'.

If π' appeals to a hypothesis, then $\phi' \in H$ and so T must be lean. Then ϕ' is of the form $x \overset{l}{\not\to}$ and ϕ of the form $x \xrightarrow{l} s$. But π is a proof of $\frac{H}{\phi}$, and so by leanness $\phi \in H$. This contradicts consistency of H.

Otherwise, the root of π' is a negative deduction step. Now, construct a provable ruloid derivation π_1 from π by replacing all subtrees concluding negative $t \overset{l}{\not\to}$ by the leaf $t \overset{l}{\not\to}$. Then π_1 is a provable ruloid derivation concluding ϕ, which denies $\iota(\phi')$. Hence, there is a formula ψ occurring in π_1 and ψ' a premise of ϕ' in π', such that ψ denies $\iota(\psi') = \psi'$. Let π_2 denote the subproof of π rooted at ψ, and π_3 the subproof of π' rooted at ψ'. But then π_2 and π_3 are proofs of denying formulae, and are smaller than π and π' respectively; by the Inductive Hypothesis, this is impossible. □

The following result shows that the set of provable formulae is closed under instantiation.

Theorem 15 (Closure under Instantiating Formulae). *Consider a formula ϕ, contexts H and K, and substitution σ. Suppose $\frac{H}{\phi}$ has a well-supported proof and that for each $\psi_i \in H$, $\frac{K}{\sigma(\psi_i)}$ has a well-supported proof. Then $\frac{K}{\sigma(\phi)}$ has a well-supported proof.*

Corollary 16 *(i) If $\overline{\phi}$ is ws-provable, then so is $\overline{\sigma(\phi)}$. (ii) If $\overline{\phi}$ is provable and ϕ is closed, then $\overline{\phi}$ has a closed well-supported proof.*

The following theorem states that our notion of well-supported proof is preserved under instantiation of deduction rules in the TSS.

Theorem 17 (Closure under Instantiating Deduction Rules). *Consider a TSS $T = (\Sigma, L, D)$ and a set of deduction rules $D' \subseteq D$; let T' be $(\Sigma, L, D \cup \{\sigma_d(d) \mid d \in D'\})$, where σ_d is an arbitrary substitution for each $d \in D'$. Then a deduction rule $\frac{H}{\phi}$ is provable with respect to T if and only it is provable with respect to T'.*

The proofs are omitted due to lack of space, but an appendix with full proofs can be found online at `www.plancomps.org/churchill2013c/`.

4 Modularity

4.1 Closed Instance Semantics

One can assign meaning to open formulae in a TSS via *closed-instance semantics*. This instantiates the deduction rules by all possible closed substitutions and considers the resulting formulae provable from the closed TSS.

Definition 18 (Closed-Instance Semantics). *Given a TSS $T = (\Sigma, L, D)$, closed(T) is defined as $(\Sigma, L, \{\sigma(d) \mid d \in D, \sigma : V \to \mathbb{C}(\Sigma)\})$, i.e., the set of deduction rules obtained by applying all closed substitutions on the deduction rules in D. The closed-instance semantics of a TSS T is the set of all closed formulae ϕ that have a ground well-supported proof with respect to closed(T).*

In such a setting, an open formula ϕ holds in T if and only if for all closed substitutions σ, $\sigma(\phi)$ has a ground well-supported proof in *closed*(T). The following example demonstrates that this does not coincide with ϕ having a well-supported proof in our setting.

Example 19 (Closed-Instance Semantics). *Consider TSS T_0 with constant 0, unary function f, labels a, b and deduction rule $\frac{x \overset{b}{\nrightarrow}}{f(x) \xrightarrow{a} x}$. For each closed term p, there is a ground well-supported proof in closed(T_0) for the deduction rule $\overline{f(p) \xrightarrow{a} p}$; hence, according to the closed-instance semantics, $\overline{f(x) \xrightarrow{a} x}$ holds. However, by Remark 8 there is no well-supported proof for $\overline{x \overset{b}{\nrightarrow}}$ in T_0, and so no well-supported proof of $\overline{f(x) \xrightarrow{a} x}$.*

For closed-instance semantics, a formula ϕ may hold in T_0 while failing in some *disjoint extension* [15] $T_0 \uplus T_1$ – closed-instance semantics is not *modular*.

Definition 20 (Disjoint Extension). *Consider two TSSs* $T_0 = (\Sigma_0, L_0, D_0)$ *and* $T_1 = (\Sigma_1, L_1, D_1)$ *of which the signatures agree on the arity of the shared function symbols. The* extension *of* T_0 *with* T_1*, denoted by* $T_0 \cup T_1$*, is defined as* $(\Sigma_0 \cup \Sigma_1, L_0 \cup L_1, D_0 \cup D_1)$*.* $T_0 \cup T_1$ *is a* disjoint *extension of* T_0 *when each deduction rule in* T_1 *is* f*-defining for some* $f \in \Sigma_1 \setminus \Sigma_0$*.*

Example 21 (Non-modularity of Closed-Instance Semantics). *Consider the TSS given in Example 19 and extend it by constant* 1 *with deduction rule* $\dfrac{}{1 \xrightarrow{b} 1}$*. Then there is no (ground) well-supported proof for* $\dfrac{}{f(1) \xrightarrow{a} 1}$ *and hence,* $\dfrac{}{f(x) \xrightarrow{a} x}$ *no longer holds for closed-instance semantics.*

4.2 Modularity for Well-Supported Proofs

In contrast, we can show that well-supported proofs are modular: well-supported proofs in T_0 remain so in $T_0 \uplus T_1$.

In the following results, by abusing the notation, we write $s \in T$ to mean s is a term in the signature of TSS T. Similarly, we write $\phi \in T$ to denote that ϕ is a formula comprising terms and labels from T. For a substitution σ, $\sigma \in T$ indicates that for all x, $\sigma(x) \in T$. We will require the following lemma for factorising substitutions:

Lemma 22. *Let* $T_0 \uplus T_1$ *be a disjoint extension of* T_0*. Let* ϕ *be a formula in* $T_0 \uplus T_1$*, and* ψ, ω *be formulae in* T_0*. Let* $\sigma, \tau \in T_0 \uplus T_1$ *be substitutions such that* $\sigma(\psi) = \tau(\omega) = \phi$*. Then there exists substitutions* $\hat{\sigma} \in T_0$*,* $\hat{\tau} \in T_0$ *and* $\rho \in T_0 \uplus T_1$ *such that* $\sigma = \rho \circ \hat{\sigma}$*,* $\tau = \rho \circ \hat{\tau}$ *and* $\hat{\sigma}(\omega) = \hat{\tau}(\psi)$*.*

We first show that each provable ruloid deduction in $T_0 \uplus T_1$ whose conclusion is an instance of a T_0-formula can be approximated by a provable ruloid deduction in T_0. We do this using the following definition of "at the root" derivation, which approximates another derivation by proving the same conclusion from a possibly richer context.

Definition 23 (At The Root Derivation). *A derivation* ϕ *is* at the root *of a derivation* ψ *if the root node of* ϕ *is the root node of* ψ*, and any immediate subproof of* ϕ *is at the root of an immediate subproof of* ψ*.*

For example, $\dfrac{x \xrightarrow{a} w}{f(x,y) \xrightarrow{b} g(w,z)}$ is at the root of $\dfrac{x \xrightarrow{a} w \quad y \xrightarrow{a} z}{f(x,y) \xrightarrow{b} g(w,z)}$.

Lemma 24 (Provable Ruloid Approximation). *Let* $T_0 \uplus T_1$ *be a disjoint extension of* T_0*. Suppose* π *is a provable ruloid derivation in* $T_0 \uplus T_1$ *concluding* ϕ *with* $\phi = \sigma(\psi)$ *for* $\sigma \in T_0 \uplus T_1$ *and* $\psi \in T_0$*. Then there exists substitutions* $\tau \in T_0$*,* $\overline{\tau} \in T_0 \uplus T_1$ *with* $\sigma = \overline{\tau} \circ \tau$*, and a provable ruloid derivation* $\pi' \in T_0$ *concluding* $\tau(\psi)$ *such that* $\overline{\tau}(\pi')$ *is at the root of* π*.*

To obtain an approximating derivation in easy: let π' consist of a single hypothesis node ψ and set $\tau = \iota$ and $\overline{\tau} = \sigma$. But this is not a provable ruloid derivation: its hypothesis ψ may be positive but not have a variable at its source. The next lemma shows that given such an approximating derivation, one can improve it. Repeated application of this lemma then yields a provable ruloid derivation.

Lemma 25. *Under the hypotheses of Lemma 24, suppose further that $\sigma = \overline{\tau} \circ \tau$ with $\tau \in T_0$ and $\pi' \in T_0$ concludes $\tau(\psi)$ with $\overline{\tau}(\pi')$ at the root of π. Suppose that π' has a positive hypothesis (leaf) whose source is not a variable. Then there exists $\tau_1 \in T_0$, $\overline{\tau}_1$ with $\sigma = \overline{\tau}_1 \circ \tau_1$ and $\pi'_1 \in T_0$ concluding $\tau_1(\psi)$ such that $\overline{\tau}_1(\pi'_1)$ is at the root of π, with π'_1 strictly larger than π'.*

Proof. By assumption, there exists a hypothesis χ in π' at position P of the form $s \xrightarrow{l} s'$ where s is not a variable. Then $\overline{\tau}(\chi) = \overline{\tau}(s \xrightarrow{l} s')$ appears in π. This cannot be a hypothesis of π, as $\overline{\tau}(s)$ is not a variable and π is a provable ruloid derivation. Hence, $\overline{\tau}(\chi)$ must appear in π as the conclusion of a deduction rule d under substitution ρ (from premises ϕ_i). Rule d must occur in T_0 since $T_0 \uplus T_1$ is a disjoint extension of T_0 and the head symbol of $\overline{\tau}(s)$ is the head symbol of s and so in T_0. Suppose $d = \dfrac{\{\omega_i : i \in I\}}{\omega}$ with $\overline{\tau}(\chi) = \rho(\omega)$ and $\phi_i = \rho(\omega_i)$. Since ω and χ are both in T_0 we may apply Lemma 22 to obtain $\hat{\tau}, \hat{\rho} \in T_0$ and $\overline{\tau}_1$ with $\overline{\tau} = \overline{\tau}_1 \circ \hat{\tau}$, $\rho = \overline{\tau}_1 \circ \hat{\rho}$ and $\hat{\tau}(\chi) = \hat{\rho}(\omega)$. Let π'_1 be $\hat{\tau}(\pi')$ attached to $\hat{\rho}(d)$ at P and let $\tau_1 = \hat{\tau} \circ \tau$. Then π'_1 concludes $\tau_1(\psi) = \hat{\tau} \circ \tau(\psi)$. Also, $\overline{\tau}_1(\pi'_1)$ is at the root of π, as $\overline{\tau}_1(\hat{\tau}(\pi')) = \overline{\tau}(\pi')$ and $\overline{\tau}_1(\hat{\rho}(\omega_i)) = \rho(\omega_i) = \phi_i$. □

Proof of Lemma 24. First, set $\tau_0 = \iota$, $\overline{\tau}_0 = \sigma$ and π'_0 the derivation consisting of a single (hypothesis) node ψ. We then repeatedly apply Lemma 25 obtaining τ_i, $\overline{\tau}_i$, π'_i until some π'_j is a provable ruloid. This process terminates, as each π_i strictly increases in size, but does not exceed the size of π. We then set $\pi' = \pi'_j$, $\tau = \tau_j$ and $\overline{\tau} = \overline{\tau}_j$. □

Using Lemma 24, we next show that well-supported proofs are preserved by disjoint extensions.

Theorem 26 (Modularity for Well-Supported Proofs). *Suppose $T_0 \uplus T_1$ is a disjoint extension of T_0 and let π be a well-supported proof (resp. derivation) for $\dfrac{H}{\phi}$ in T_0. Then π is a well-supported proof (resp. derivation) for $\dfrac{H}{\phi}$ in $T_0 \uplus T_1$.*

Proof. We first consider derivations. Each derivation in T_0 is also one in $T_0 \uplus T_1$. This follows from a straightforward induction, as each deduction rule in T_0 is also a deduction rule in $T_0 \uplus T_1$.

We now consider the case for well-supported proofs. We proceed by induction on π. If the derivation just appeals to a hypothesis, then it is also valid in $T_0 \uplus T_1$. If the root formula ϕ is positive and the derivation applies an instance of a deduction rule of T_0 to obtain sub-derivations $\{\pi_i : i \in I\}$ above ϕ, then we may apply the inductive hypothesis to the nodes above ϕ and apply the same instance of the deduction rule to see that π is a proof in $T_0 \uplus T_1$.

If ϕ is negative and π has root $\frac{\{\psi_i : i \in I\}}{s \overset{l}{\nrightarrow}}$, then we must show that for each provable ruloid derivation $\pi' \in T_0 \uplus T_1$ concluding $\sigma(s) \overset{l}{\longrightarrow} s'$, there is a formula occurring in π' denying some $\sigma(\psi_i)$. Consider such a π' and fresh x occurring in no ψ_i, and let $\sigma' = \sigma[x \mapsto s']$. Then π' concludes $\sigma'(s \overset{l}{\longrightarrow} x)$. Since $s \overset{l}{\longrightarrow} x$ is a formula in T_0, we may apply Lemma 24 to construct τ, $\overline{\tau}$ and π'' as described with $\overline{\tau}(\pi'')$ at the root of π'. Derivation π'' is in T_0 and concludes $\tau(s \overset{l}{\longrightarrow} x)$, which denies $\tau(s \overset{l}{\nrightarrow})$. Since π is a well-supported proof there is a formula ψ' occurring in π'' denying some $\tau(\psi_i)$. Then $\overline{\tau}(\psi')$ occurs in $\overline{\tau}(\pi'')$ and so in π', and denies $\overline{\tau}(\tau(\psi_i)) = \sigma'(\psi_i) = \sigma(\psi_i)$, as required. □

5 Conservativeness

5.1 Conservativeness for Disjoint Extensions

We next show that for *source-dependent* TSSs, a disjoint extension of a TSS does not introduce additional provable formulae from the original TSS. In [10], an analogous result is presented for closed terms in the more abstract setting of three-valued stable models. From there, we recall the notion of *source-dependency*:

Definition 27 (Source-Dependency). *Given a proof rule, the* source-dependent variables *are defined inductively as follows:*

- *All variables in the source of the conclusion are source-dependent.*
- *If all variables in the source of a premise are source-dependent, so are those in the conclusion of that premise.*

A rule is source-dependent *if all variables it mentions are. A TSS is* source-dependent *if all of its rules are.*

Theorem 28 (Conservativeness for Disjoint Extensions). *Let $T_0 \uplus T_1$ be a disjoint extension of T_0, where T_0 is source-dependent, and let $\phi \in T_0$. Let π be a well-supported proof (resp. derivation) for $\frac{H}{\phi}$ in $T_0 \uplus T_1$. Then π is a well-supported proof (resp. derivation) for $\frac{H}{\phi}$ in T_0.*

Proof. (Sketch) For derivations and positive steps in well-supported proofs, we proceed by an outer induction on the proof and an inner induction on the source-dependence measure. For negative steps in well-supported proofs, we can use Theorem 26 to see that any denying derivation in T_0 is also one in $T_0 \uplus T_1$. □

The following example demonstrates why source-dependency is necessary for the above result (it is violated by the occurrence of x):

Example 29. *Consider a TSS T_0 with constants 0 and 1, labels a and b, and rule $\frac{x \overset{b}{\longrightarrow} 1}{0 \overset{a}{\longrightarrow} 1}$. Let $T_0 \uplus T_1$ extend T_0 with constant 2 and rule $2 \overset{b}{\longrightarrow} 1$. Then $\frac{}{0 \overset{a}{\longrightarrow} 1}$ is provable in $T_0 \uplus T_1$ but not in T_0, while $0 \overset{a}{\longrightarrow} 1$ is a formula of T_0.*

5.2 Conservativeness over Closed-Instance Semantics

We next consider how our notion of well-supported-proof relates to the original notion of ground well-supported proof [12]. We first show that if a closed formula has a well-supported proof in T, then it has a ground well-supported proof in $\mathit{closed}(T)$. To do this, we define the notion of strict proof, which requires that the premises of a negative formula may not involve negative non-GSOS formulae.

Definition 30 (Strict Well-Supported Proof). *A* strict well-supported proof *is one in which if a negative formula ϕ occurs above a negative formula ψ then the source of ϕ is a variable.*

Lemma 31. *If $\frac{\Gamma}{\phi}$ has a (closed) well-supported proof, then it has a strict (closed) well-supported proof.*

Theorem 32 (Soundness w.r.t. ground well-supported proofs). *For each closed formula ϕ, if $\overline{\phi}$ has a well-supported proof in T, then $\overline{\phi}$ has a ground well-supported proof in $\mathit{closed}(T)$.*

Proof. If $\overline{\phi}$ has a well-supported proof, then by Corollary 16 it has a closed well-supported proof, and by Lemma 31 a strict closed well-supported proof π. We claim that π is a ground well-supported proof of $\overline{\phi}$ in $\mathit{closed}(T)$.

Each positive step in π is a closed instance of a deduction rule in T. This is a valid step in a ground well-supported proof in $\mathit{closed}(T)$.

For the negative case, suppose the root is $\frac{K}{\phi}$. Let π' witness provable rule $\frac{H}{\phi'}$ in $\mathit{closed}(T)$ where ϕ' denies ϕ. Then π' is also a provable ruloid derivation in T, concluding ϕ' which denies $\iota(\phi)$. Since π is a well-supported proof in T, there is some $\chi \in K$ and χ' occurring in π' where χ' denies $\iota(\chi) = \chi$. Since χ' occurs in π' it must be closed, and so the source of χ must be closed. Since the source of χ is not a variable, strictness of π ensures that it is positive, and χ' negative. Since negative χ' occurs in provable ruloid derivation π', it must occur as a leaf, with $\chi' \in H$. Thus we have found $\chi \in K$ and $\chi' \in H$ denying χ, as required. □

For the converse, we will require source-dependency. The following example show that without source-dependency, the converse implication does not hold.

Example 33. *Consider TSS T with constants 0 and 1, labels a and b, and deduction rule $\frac{x \xrightarrow{b} 1}{0 \xrightarrow{a} 1}$. In $\mathit{closed}(T)$, $\overline{0 \xrightarrow{a}}$ has a ground well-supported proof as there are no provable rules concluding $0 \xrightarrow{a} s$. But it does not have a well-supported proof in T: the provable ruloid derivation $\frac{x \xrightarrow{b} 1}{0 \xrightarrow{a} 1}$ would require a well-supported proof of $\overline{x \xrightarrow{b}} 1$, which does not exist by Remark 8.*

The following proposition and subsequent theorem show that in source-dependent systems the converse of Theorem 32 holds.

Proposition 34 *Consider a source-dependent TSS T. Let ϕ be a formula whose source is closed and let π be a derivation in T concluding ϕ. Then π is a derivation in closed(T).*

Theorem 35 (Conservativeness over Closed-Instance Semantics). *Consider a source-dependent TSS T. For each closed formula ϕ, if $\overline{\phi}$ has a ground well-supported proof with respect to closed(T), then $\overline{\phi}$ has a well-supported proof with respect to T.*

Proof. Let π be the derivation in *closed*(T) witnessing $\overline{\phi}$. We show that π is also a well-supported proof in T. Since there are no hypotheses to appeal to, the only cases we need to consider are the positive and negative deduction steps. For the positive steps, any instance of a proof rule in *closed*(T) is an instance of a proof rule in T.

For the negative case, suppose $\frac{K}{\phi}$ occurs in π with ϕ negative. Let π' witness a provable ruloid $\frac{H}{\phi'}$ where ϕ' denies $\sigma(\phi)$. Then ϕ is closed since π is a deduction in *closed*(T), and so $\sigma(\phi)$ and the source of ϕ' are closed. By Proposition 34, π' is a derivation in *closed*(T). Each leaf of π' is in context H and closed, so must be a negative formula. Thus π' witnesses the provable rule $\frac{H}{\phi'}$. Since ϕ' denies $\sigma(\phi)$ and the source of ϕ is closed, ϕ' denies ϕ. Since π is a ground well-supported proof, there is a hypothesis $\chi \in K$ with negative $\chi' \in H$ where χ' denies χ. Since χ is closed, χ' also denies $\sigma(\chi)$. Thus χ' occurs in π' and denies $\sigma(\chi)$ with $\chi \in H$, as required. □

6 Conclusions

In this paper, we introduced a notion of semantics for open terms with respect to transition system specifications with negative premises. This notion extends the traditional notions [6,12] (which were confined to closed terms) and enjoys a number of intuitive properties: consistency, closure under instantiation, modularity and conservativeness. Consistency means that no two denying formulae are provable. Closure under instantiation means that firstly, instantiating deduction rules does not change the set of provable formulae and secondly, the set of provable formulae is closed under applying substitutions. Modularity means that all provable open formulae remain provable under disjoint extensions of the transition system specification. Conservativeness means that firstly, disjoint extensions do not introduce new provable formulae from the original TSS and secondly, our notion of semantics leads to the same set of provable closed transition formulae as the traditional notion.

This research was initiated by our study of bisimulation for open terms, in particular with regards to congruence (compositionality) and preservation under disjoint extensions (modularity). Earlier results consider open notions of bisimilarity purely positive TSSs (e.g., [7,15,1,18,19]). We hope to use the results here to extend these results to the systems with negative premises (such as those in the (n)tyft/(n)tyxt [13], ntree [9] or PANTH [20] formats).

Acknowledgements. Many thanks to the anonymous referees for their useful comments. This work was supported by an EPSRC grant (EP/I032495/1) to Swansea University in connection with the *PLanCompS* project (`www.plancomps.org`).

References

1. Aceto, L., Cimini, M., Ingólfsdóttir, A.: Proving the validity of equations in GSOS languages using rule-matching bisimilarity. MSCS 22(2), 291–331 (2012)
2. Aceto, L., Fokkink, W.J., Verhoef, C.: Structural operational semantics. In: Handbook of Process Algebra, ch. 3, pp. 197–292. Elsevier (2001)
3. Aceto, L., Ingolfsdottir, A.: On the expressibility of priority. Inf. Process. Lett. 109(1), 83–85 (2008)
4. Bloom, B., Istrail, S., Meyer, A.R.: Bisimulation can't be traced. JACM 42(1), 232–268 (1995)
5. Bloom, B., Fokkink, W., van Glabbeek, R.J.: Precongruence formats for decorated trace semantics. ACM Trans. Comput. Logic 5(1), 26–78 (2004)
6. Bol, R., Groote, J.F.: The meaning of negative premises in transition system specifications. JACM 43(5), 863–914 (1996)
7. Churchill, M., Mosses, P.D.: Modular bisimulation theory for computations and values. In: Pfenning, F. (ed.) FOSSACS 2013. LNCS, vol. 7794, pp. 97–112. Springer, Heidelberg (2013)
8. Clark, K.L.: Negation as failure. In: Proc. ADBT 1977, pp. 293–322. Plemum Press (1978)
9. Fokkink, W.J., van Glabbeek, R.J.: Ntyft/ntyxt rules reduce to ntree rules. I&C 126(1), 1–10 (1996)
10. Fokkink, W.J., Verhoef, C.: A conservative look at operational semantics with variable binding. I&C 146(1), 24–54 (1998)
11. van Glabbeek, R.J.: The meaning of negative premises in transition system specifications II. Tech. Report, Stanford (STAN-CS-TN-95-16) (1995)
12. van Glabbeek, R.J.: The meaning of negative premises in transition system specifications II. JLAP 60-61, 229–258 (2004)
13. Groote, J.F.: Transition system specifications with negative premises. TCS 118(2), 263–299 (1993)
14. Groote, J.F., Vaandrager, F.W.: Structured operational semantics and bisimulation as a congruence. I&C 100(2), 202–260 (1992)
15. Mosses, P.D., Mousavi, M.R., Reniers, M.A.: Robustness of equations under operational extensions. In: Proc. EXPRESS 2010. EPTCS, vol. 41, pp. 106–120 (2010)
16. Mousavi, M., Reniers, M.A., Groote, J.F.: SOS rule formats and meta-theory: 20 years after. TCS 373, 238–272 (2007)
17. Plotkin, G.D.: A structural approach to operational semantics. JLAP 60-61, 17–139 (2004)
18. Rensink, A.: Bisimilarity of open terms. I&C 156, 345–385 (2000)
19. de Simone, R.: Higher-level synchronizing devices in MEIJE-SCCS. TCS 37, 245–267 (1985)
20. Verhoef, C.: A congruence theorem for structured operational semantics with predicates and negative premises. Nord. J. of Comp. 2(2), 274–302 (1995)

Mutually Testing Processes*
(Extended Abstract)

Giovanni Bernardi and Matthew Hennessy

School of Computer Science and Statistics,
Trinity College Dublin,
Dublin 2, Ireland
bernargi@tcd.ie, matthew.hennessy@cs.tcd.ie

Abstract. In the standard testing theory of DeNicola-Hennessy one process is considered to be a refinement of another if every test guaranteed by the former is also guaranteed by the latter. In the domain of web services this has been recast, with processes viewed as *servers* and tests as *clients*. In this way the standard refinement preorder between servers is determined by their ability to satisfy clients.

But in this setting there is also a natural refinement preorder between *clients*, determined by their ability to be satisfied by *servers*. In more general settings where there is no distinction between *clients* and *servers*, but all processes are *peers*, there is a further refinement preorder based on the mutual satisfaction of *peers*.

We give a uniform account of these three preorders. In particular we give two characterisations. The first is behavioural, in terms of traces and ready sets. The second, for finite processes, is equational.

1 Introduction

The DeNicola-Hennessy theory of testing [NH84, DH87, Hen88] considers a process p to be a refinement of process q if every test passed by p is also passed by q. Recently, in papers such as [LP07, Bd10, CGP09, Pad10], this refinement preorder has been recast with a view to providing theoretical foundations for web services. Here processes are viewed as *servers* and tests viewed as *clients*. In this terminology the standard (must) testing preorder is a refinement preorder between servers, which we denote by $p \sqsubseteq_{\mathsf{svr}} q$; this is determined by the ability of the servers p, q to satisfy clients. However in this framework there are many other natural behavioural preorders between processes. In this paper we investigate two; the first, $p \sqsubseteq_{\mathsf{clt}} q$, is determined by the ability of the clients p, q to be satisfied by servers. For the second we drop the distinction between clients and servers. Instead all processes are viewed as peers of each other and the purpose of interaction between two peers is the mutual satisfaction of both. The resulting refinement preorder is denoted by $p \sqsubseteq_{\mathsf{p2p}} q$. We give a uniform behavioural characterisation of all three refinement preorders in terms of traces and *acceptances*

* Research supported by SFI project SFI 06 IN.1 1898.

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 61–75, 2013.

sets [NH84, Hen88]. We also give equational characterisations for a finite process calculus for servers/clients/peers.

We use an infinitary version of CCS [Mil89] augmented by a *success* constant $\mathsf{1}$, to describe processes, be they servers, clients or peers. Thus $p = \tau.a.(b.\mathsf{0} + c.\mathsf{0}) + \tau.a.c.\mathsf{0}$ is a server which offers the action a followed by either b and c depending on how choices are made, and then terminates, denoted by $\mathsf{0}$. On the other hand $r = \overline{a}.\overline{c}.\mathsf{1}$ is a test or a client which seeks a synchronisation on a followed by one on c; as usual [Mil89] communication or cooperation consists of the simultaneous occurrence of an action a and its complement $\overline{a}$. Thus when the server p is executed in parallel with the client r, the latter will always be satisfied, in that it is guaranteed to reach the successful state $\mathsf{1}$ regardless of how the various choices are made. But if the client is executed with the alternative server $q = \tau.a.b.\mathsf{0} + \tau.a.c.\mathsf{0}$ there is a possibility of the client remaining unhappy; for this reason $p \not\sqsubseteq_{\mathsf{svr}} q$. However it turns out that $q \sqsubseteq_{\mathsf{svr}} p$ because every client satisfied by q will also be satisfied by p.

The client preorder $p \sqsubseteq_{\mathsf{clt}} q$ compares the processes as clients, and their ability to be satisfied by servers. This refinement preorder turns out to be incomparable with the server preorder. For example $a.\mathsf{1} + b.\mathsf{0} \not\sqsubseteq_{\mathsf{svr}} a.\mathsf{1}$ because of the client $\overline{b}.\mathsf{1}$. But $a.\mathsf{1} + b.\mathsf{0} \sqsubseteq_{\mathsf{clt}} a.\mathsf{1}$ because every server satisfying the former also satisfies $a.\mathsf{1}$; intuitively the extra component of the client $b.\mathsf{0}$ puts no further demands on servers, because the execution of b will never lead to satisfaction. Conversely $a.\mathsf{1} \sqsubseteq_{\mathsf{svr}} a.\mathsf{0}$ because $\mathsf{1}$ plays no role for processes acting a servers, while $a.\mathsf{1} \not\sqsubseteq_{\mathsf{clt}} a.\mathsf{0}$; $a.\mathsf{1}$ as a client is satisfied by the server $\overline{a}.\mathsf{0}$ while $a.\mathsf{0}$ can never be satisfied as a client by any server. Behaviour relative to the client preorder $\sqsubseteq_{\mathsf{clt}}$ is very sensitive to the presence of $\mathsf{1}$ and $\mathsf{0}$; for example $\mathsf{0}$ is a least element, that is $\mathsf{0} \sqsubseteq_{\mathsf{clt}} r$ for any process r.[1] However in general the precise role these constants play is difficult to discern; for example, rather surprisingly we have $a.(b.\mathsf{0} + c.\mathsf{1}) + a.(b.\mathsf{1} + c.\mathsf{0}) \sqsubseteq_{\mathsf{clt}} \mathsf{0}$.

If we ignore the distinction between servers and clients then every process plays an independent role as a *peer* to all other processes in its environment. This point of view leads to another behavioural preorder. Intuitively, we say that the process p satisfies its peer q if whenever they are executed in parallel both are guaranteed to be satisfied; in some sense both peers test their partner. Then $p_1 \sqsubseteq_{\mathsf{p2p}} p_2$ means that every peer satisfied by p_1 is also satisfied by p_2.

This third refinement preorder is different from the server and client preorders. In fact we will show that $p_1 \sqsubseteq_{\mathsf{p2p}} p_2$ implies $p_1 \sqsubseteq_{\mathsf{clt}} p_2$; but the converse is not true in general. For example $\mathsf{1} + b.\mathsf{0} \sqsubseteq_{\mathsf{clt}} \mathsf{1}$ but $\mathsf{1} + b.\mathsf{0} \not\sqsubseteq_{\mathsf{p2p}} \mathsf{1}$ because of the peer $b.\mathsf{1}$. In our formulation $\mathsf{1} + b.\mathsf{0}$ and $\overline{b}.\mathsf{1}$ mutually satisfy each other, whereas the peers $\mathsf{1}$ and $b.\mathsf{1}$ do not.

The aim of the paper is to show that the theory of the standard (must) testing preorder [NH84, Hen88], here formulated as the server refinement preorder $\sqsubseteq_{\mathsf{svr}}$, can be extended to both the client and the peer refinement preorders.

[1] Note in passing that this is not the case for the server preorder; $\mathsf{0}$ as a server guarantees the client $\overline{b}.\mathsf{0} + \tau.\mathsf{1}$ but the server $b.\mathsf{0}$ does not.

It is well-known that the behaviour of processes relative to $\sqsubseteq_{\mathsf{svr}}$ can be characterised in terms of the traces they can perform followed by *ready* or *acceptance* sets; intuitively each ready set A after a trace s captures a possibility for the process to deadlock when interacting with a client. For example the process $q = \tau.a.b.\,0 + \tau.a.c.\,0$ has the ready set $\{\, b \,\}$ after the (weak) sequence of actions a; this represents the possibility of q deadlocking if servicing a client which requests an action a but then is not subsequently interested in the action b. The process $p = a.(b.\,0 + c.\,0) + a.c.\,0$, also discussed above, has no comparable ready set and for this reason $p \not\sqsubseteq_{\mathsf{svr}} q$.

The first main result of the paper is a similar behavioural characterisation of both the client and the peer refinement preorders, in terms of certain kinds of traces and ready sets. However the details are intricate. It turns out that *unsuccessful* traces, those which can be performed without reaching a successful state, play an essential role. We also need to parametrise these concepts, relative to *usable* actions and *usable* processes; the exact meaning of *usable* will depend on the particular refinement preorder being considered.

It is also well-known that the standard testing preorders over finite processes can be characterised by a collection of (in-)equations over the process operators, [NH84, Hen88]. The second main result of the paper is a similar characterisation of the new refinement preorders. In fact there is a complication here, as these preorders are not in general preserved by the external operator $+$. A similar complication occurred in Section 7.2 of [Mil89] in the axiomatisation of *weak bisimulation equivalence*, and in the axiomatisations of the *must testing* preorder in [NH84], and we adopt the same solution. We give sound and complete (in-)equational theories for the largest pre-congruences $\sqsubseteq^{\mathsf{c}}_{\mathsf{clt}}$, $\sqsubseteq^{\mathsf{c}}_{\mathsf{p2p}}$ contained in the refinement preorders $\sqsubseteq_{\mathsf{clt}}$, $\sqsubseteq_{\mathsf{p2p}}$ respectively, over a finite version of CCS. The presence of the success constant 1 in this language complicates the axiomatisations considerably, as the behaviour of clients and peers is very dependent on their ability to immediately report success. For this reason we reformulate the axiomatisation of *must testing* preorder from [NH84], which in this paper coincides with the server preorder $\sqsubseteq^{\mathsf{c}}_{\mathsf{svr}}$, as a two-sorted equational theory. The characterisation of the client and server preorders, $\sqsubseteq^{\mathsf{c}}_{\mathsf{clt}}$, $\sqsubseteq^{\mathsf{c}}_{\mathsf{svr}}$ respectively, requires extra equations to capture the behaviour of the special processes 1 and 0. For example one of the inequations required by the client preorder is $x \leq 1$, while those for the peer preorder include $\mu.(1 + x) \leq 1 + \mu.x$.

The remainder of this extended abstract is organised as follows. Section 2 is devoted to definitions and notation. We introduce a language for describing processes, an infinitary version of the CCS used in [Mil89], and give the standard intensional interpretation of it as a labelled transition system, LTS. For the remainder of the paper, processes will then be considered to be states in the resulting LTS. We also formally define the three different refinement preorders discussed informally in the Introduction, by generalising the standard notion from [NH84] of applying tests to processes. We begin Section 3 by recalling the well-known characterisation of the must preorder (Theorem 1) for finite branching LTSs from [NH84] in terms of traces and ready sets. To adapt this for

the client preorder we need some extra technical notation. This is motivated by a series of examples, until we finally obtain a statement of the characterisation (Theorem 2).

The notation used in this characterisation of the client preorder can be modified in a uniform manner to give an analogous characterisation of the server preorder, (Theorem 3), which applies even in LTSs which are not finite-branching. Finally by combining these we get an analogous characterisation (Theorem 5) for the peer preorder.

In Section 4 we restrict our attention to a finite sub-language $\mathsf{CCS}^{\mathrm{f}}$ and address the question of equational characterisations. We first show why the client and peer refinement preorders are not preserved by the external choice operator $+$, and give a simple behavioural characterisation of the associated pre-congruences $\sqsubseteq^{\mathrm{c}}_{\mathsf{svr}}$, $\sqsubseteq^{\mathrm{c}}_{\mathsf{clt}}$ and $\sqsubseteq^{\mathrm{c}}_{\mathsf{p2p}}$; this simply involves taking into account the initial behaviour of processes. We then explain the equations which need to be added to the standard set in order to obtain completeness (Theorem 9 and Theorem 8). The paper ends with Section 5, where we present a summary of our results, a comparison with the existing work, and a series of open questions.

In this extended abstract all proofs are omitted. The proofs of the various behavioural characterisations from Section 3 will appear in [Ber13].

2 Testing Processes

Let Act be a set of actions, ranged over by $a, b, c, \ldots$ and let τ, $\checkmark$ be two distinct actions *not* in Act; the first will denote internal unobservable activity while the second will be used to report the success of an experiment. To emphasise their distinctness we use Act_τ to denote the set $\mathsf{Act} \cup \{\,\tau\,\}$, and similarly for $\mathsf{Act}_{\tau\checkmark}$; we use μ to range over the former and λ to range over the latter. We assume Act has an idempotent complementation function, with $\overline{a}$ being the complement to a. A labelled transition system, LTS, consists of a triple $\langle\, P, \mathsf{Act}_{\tau\checkmark}, \longrightarrow \rangle$, where P is a set of processes and $\longrightarrow \subseteq P \times \mathsf{Act}_{\tau\checkmark} \times P$ is a transition relation between processes decorated with labels drawn from the set $\mathsf{Act}_{\tau\checkmark}$. We use the infix notation $p \xrightarrow{\lambda} q$ in place of $(p, \lambda, q) \in \longrightarrow$. An LTS is finite-branching if for all $p \in P$ and for all $\lambda \in \mathsf{Act}_{\tau\checkmark}$, the set $\{\, q \mid p \xrightarrow{\lambda} q \,\}$ is finite. Single transitions $p \xrightarrow{\lambda} q$ are extended to sequences of transitions $p \xrightarrow{t} q$, where $t \in (\mathsf{Act}_{\tau\checkmark})^\star$, in the standard manner. For $s \in (\mathsf{Act}_\checkmark)^\star$ we also have the standard weak transitions, $p \stackrel{s}{\Longrightarrow} q$, defined by ignoring the occurrences of τs. Somewhat nonstandard is the use of infinite weak transitions, $p \stackrel{u}{\Longrightarrow}$, for $u \in (\mathsf{Act})^\infty$.

It will be convenient to have a notation for describing LTSs; we use an infinitary version of CCS, [Mil89], augmented with a *success* operator, $\mathsf{1}$. The syntax of the language is depicted in Figure 1. We use $\mathsf{0}$ to denote the empty external sum $\sum_{i\in\emptyset} p_i$ and $p_1 + p_2$ for the binary sum $\sum_{i\in\{\,1,2\,\}} p_i$. If I is a non-empty set, we use $\bigoplus_{i\in I} p_i$ to denote the sum $\sum_{i\in I} \tau.p_i$. For the remainder of the paper we use the LTS whose states are the terms in CCS and where the relations $p \xrightarrow{\lambda} q$ are the least ones determined by the (standard) rules in Figure 2. We use *finite*

$$p, q, r ::= 1 \;\mid\; A \;\mid\; \mu.p \;\mid\; \sum_{i \in I} p_i \;\mid\; p \,||\, q$$

where I is a countable index set, and A ranges over a set of definitional constants each of which has an associated definition $A \stackrel{\text{def}}{=} p_A$.

Fig. 1. Syntax of infinitary CCS

$$\frac{}{1 \xrightarrow{\checkmark} 0}\ (\text{A-Ok}) \qquad \frac{}{\mu.p \xrightarrow{\mu} p}\ (\text{A-Pre})$$

$$\frac{p \xrightarrow{\lambda} p'}{p + q \xrightarrow{\lambda} p'}\ (\text{R-Ext-L}) \qquad \frac{q \xrightarrow{\lambda} q'}{p + q \xrightarrow{\lambda} q'}\ (\text{R-Ext-R})$$

$$\frac{q \xrightarrow{\lambda} q'}{q \,||\, p \xrightarrow{\lambda} q' \,||\, p}\ (\text{P-Left}) \qquad \frac{p \xrightarrow{\lambda} p'}{q \,||\, p \xrightarrow{\lambda} q \,||\, p'}\ (\text{P-Right})$$

$$\frac{q \xrightarrow{a} q' \quad p \xrightarrow{\overline{a}} p'}{q \,||\, p \xrightarrow{\tau} q' \,||\, p'}\ (\text{P-Synch}) \qquad \frac{p \xrightarrow{\lambda} p'}{A \xrightarrow{\lambda} p'}\ A \stackrel{\text{def}}{=} p;\ (\text{R-Const})$$

Fig. 2. The operational semantics of CCS

branching CCS to refer to the LTS which consists only of terms from CCS which generate finite branching structures.

A *computation* consists of series of τ actions of the form

$$p \,||\, r = p_0 \,||\, r_0 \xrightarrow{\tau} p_1 \,||\, r_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} p_k \,||\, r_k \xrightarrow{\tau} \ldots \tag{1}$$

It is *maximal* if it is infinite, or whenever $p_n \,||\, r_n$ is the last state then $p_n \,||\, r_n \stackrel{\tau}{\not\longrightarrow}$. A computation may be viewed as two processes p, r, one a server and the other a client, co-operating to achieve individual goals, which may or may not be independent. We say (1) is *client-successful* if there exists some $k \geq 0$ such that $r_k \xrightarrow{\checkmark}$. It is *successful* if it is *client-successful* and there exists an $l \geq 0$ such that $p_l \xrightarrow{\checkmark}$. In a *client-successful* computation the client can report *success* while in a *successful* one both the client and the server can report success; note however that they are not required to do so at the same time.

Definition 1 (Passing tests). *We write* p must r *if every maximal computation from* $p \,||\, r$ *is* client-successful. *We write* p must^{p2p} r *if every such computation is* successful. □

Intuitively, p must r means that the client r is satisfied by the server p, as r always reaches a state where it can report success. On the other hand, p must^{p2p} r

means that p passes r *and* r also passes p; so p and r *have to collaborate* in order to pass each other. Thus, when using the testing relation $\mathsf{must}^{\mathsf{p2p}}$ we think of p and r as two peers rather than a server and a client.

Definition 2 (Testing preorders). *In an arbitrary LTS we write*

(1) $p_1 \sqsubseteq_{\mathsf{svr}} p_2$ *if for every* r, p_1 must r *implies* p_2 must r
(2) $r_1 \sqsubseteq_{\mathsf{clt}} r_2$ *if for every* p, p must r_1 *implies* p must r_2
(3) $p \sqsubseteq_{\mathsf{p2p}} q$ *if for every* r, p $\mathsf{must}^{\mathsf{p2p}}$ r *implies* q $\mathsf{must}^{\mathsf{p2p}}$ r.

We use the obvious notation for the kernel of these preorders; for instance $p_1 \eqsim_{\mathsf{p2p}} p_2$ *means that* $p_1 \sqsubseteq_{\mathsf{p2p}} p_2$ *and* $p_2 \sqsubseteq_{\mathsf{p2p}} p_1$. □

The preorder $\sqsubseteq_{\mathsf{svr}}$ is meant to compare servers, as $p_1 \sqsubseteq_{\mathsf{svr}} p_2$ ensures that all the clients passed (wrt must) by p_1 are passed also by p_2. The preorder $\sqsubseteq_{\mathsf{clt}}$ relates processes seen as clients, because $r_1 \sqsubseteq_{\mathsf{clt}} r_2$ means that all the servers that satisfy r_1 satisfy also r_2. The third preorder, $\sqsubseteq_{\mathsf{p2p}}$, relates processes seen as *peers*; this follows from the fact that p $\mathsf{must}^{\mathsf{p2p}}$ r is true only if p and r mutually satisfy each other.

3 Semantic Characterisations

The standard (must) testing preorder from [NH84, Hen88] has been characterised for finite-branching LTSs using two behavioural predicates. The first, $p \Downarrow s$, says that p can never come across a divergent residual while executing the sequence of actions $s \in \mathsf{Act}^\star$. We use the notation $p \Downarrow$, p *converges*, to mean that there is no infinite sequence $p \xrightarrow{\tau} p_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} p_k \xrightarrow{\tau} \ldots$. Then the general convergence predicate is defined inductively as follows:

(a) $p \Downarrow \varepsilon$ whenever $p \Downarrow$
(b) $p \Downarrow a.s$ whenever $p \Downarrow$ and $p \stackrel{a}{\Longrightarrow}$ implies $\bigoplus(p \text{ after } a) \Downarrow s$

where $(p \text{ after } s)$ denotes the set $\{\, p' \mid p \stackrel{s}{\Longrightarrow} p' \,\}$. Note that $p \stackrel{a}{\Longrightarrow}$ ensures that $(p \text{ after } a)$ is non-empty; thus $\bigoplus(p \text{ after } a)$ represents a (well-formed) process consisting of the choice between the elements of the non-empty set $(p \text{ after } a)$, which may in general be infinite. The second predicate codifies the possible deadlocks which may occur when a process p attempts to execute the trace of actions $s \in \mathsf{Act}^\star$:

$$\mathsf{Acc}(p, s) = \{\, S(q) \mid p \stackrel{s}{\Longrightarrow} q \stackrel{\tau}{\not\longrightarrow} \,\} \tag{2}$$

where $S(q) = \{\, a \in \mathsf{Act} \mid q \xrightarrow{a} \,\}$. The sets $S(q)$ are called *ready sets*, while we say that $\mathsf{Acc}(p, s)$ is the *acceptance set* of p *after* a trace s. Ready sets are essentially the complements of the *refusal sets* used in [Hoa85]. The sets in $\mathsf{Acc}(p, s)$ describe the interactions that can lead p out of a possible deadlock, reached by executing the trace s of external actions.

Theorem 1. [DH87, Hen88] In finite branching CCS, $p \sqsubseteq_{\mathsf{svr}} q$ if and only if, for every $s \in \mathsf{Act}^\star$, if $p \Downarrow s$ then (i) $q \Downarrow s$, (ii) for every $B \in \mathsf{Acc}(q,s)$ there exists some $A \in \mathsf{Acc}(p,s)$ such that $A \subseteq B$, and (iii) if $q \stackrel{s}{\Longrightarrow}$ then $p \stackrel{s}{\Longrightarrow}$. □

As might be expected, this behavioural characterisation does not work for $\sqsubseteq_{\mathsf{clt}}$:

Example 1. One can prove that $b.a.\,1 \sqsubseteq_{\mathsf{clt}} q$, where q denotes $b.(c.\,0 + 1)$. However their acceptance sets are not related as required by Theorem 1. Calculations show that $\mathsf{Acc}(b.a.\,1, b) = \{\{a\}\}$. But $\{c\} \in \mathsf{Acc}(q,b)$ and so there is no set B in $\mathsf{Acc}(b.a.\,1, b)$ satisfying $B \subseteq \{c\}$. □

In Example 1 we should not require the ready set $\{c\} \in \mathsf{Acc}(q,b)$ to be matched by one in $\mathsf{Acc}(b.a.\,1, b)$ because q can report success immediately after performing b. We formalise this intuition. For every $s \in \mathsf{Act}^\star$ let $p \stackrel{s}{\Longrightarrow}_{\not\checkmark} q$ be the least relation satisfying

(1) $p \stackrel{\checkmark}{\not\longrightarrow}$ implies $p \stackrel{\varepsilon}{\Longrightarrow}_{\not\checkmark} p$

(2) if $p' \stackrel{s}{\Longrightarrow}_{\not\checkmark} q$ and $p \stackrel{\checkmark}{\not\longrightarrow}$ then
- $p \stackrel{a}{\longrightarrow} p'$ implies $p \stackrel{as}{\Longrightarrow}_{\not\checkmark} q$
- $p \stackrel{\tau}{\longrightarrow} p'$ implies $p \stackrel{s}{\Longrightarrow}_{\not\checkmark} q$

Intuitively, $p \stackrel{s}{\Longrightarrow}_{\not\checkmark} q$ means that p can perform the sequence of external actions s ending up in state q without passing through any state which can report success; in particular neither p nor q can report success. This notation is extended to infinite traces, $u \in \mathsf{Act}^\infty$, by letting $p \stackrel{u}{\Longrightarrow}_{\not\checkmark}$ whenever there exists a $t \in (\mathsf{Act}_\tau)^\infty$ such that $t = \mu_1\mu_2\ldots$, (a) $p = p_0 \stackrel{\mu_1}{\longrightarrow} p_1 \stackrel{\mu_2}{\longrightarrow} p_2 \stackrel{\mu_3}{\longrightarrow} \ldots$ implies that $p_i \stackrel{\checkmark}{\not\longrightarrow}$ for every p_i, and (b) for every $n \in \mathbb{N}$ and some $k \in \mathbb{N}$, $u_n = \langle t_k \rangle_{\setminus\tau}$; where $\langle t \rangle_{\setminus\tau}$ removes the τs from the string t.

Definition 3. *For every process p and trace $s \in \mathsf{Act}^\star$, let*

$$\mathsf{Acc}_{\not\checkmark}(p,s) = \{\, S(q) \mid p \stackrel{s}{\Longrightarrow}_{\not\checkmark} q \stackrel{\tau}{\not\longrightarrow} \,\}$$

We call the set $\mathsf{Acc}_{\not\checkmark}(p,s)$ the unsuccessful *acceptance set of p after s.* □

We can now try to adapt the characterisation for servers in Theorem 1 to clients as follows:

Definition 4. *Let $r_1 \preccurlyeq_{\mathsf{bad}} r_2$ if for every $s \in \mathsf{Act}^\star$, if $r_1 \Downarrow s$ then (i) $r_2 \Downarrow s$, and (ii) for every $B \in \mathsf{Acc}_{\not\checkmark}(r_2,s)$, there exists some $A \in \mathsf{Acc}_{\not\checkmark}(r_1,s)$ such that $A \subseteq B$.* □

Example 2. One can show that $r \sqsubseteq_{\mathsf{clt}} c.a.\,1$ where r denotes the client $c.(a.\,1 + b.\,0)$. However they are not related by the proposed $\preccurlyeq_{\mathsf{bad}}$ in Definition 4. Obviously $r \Downarrow c$ and $\{a\} \in \mathsf{Acc}_{\not\checkmark}(c.a.\,1, c)$. But there is no $B \in \mathsf{Acc}_{\not\checkmark}(r,c)$ such that $B \subseteq \{a\}$; this is because $\mathsf{Acc}_{\not\checkmark}(r,c) = \{\{a,b\}\}$. The problem is the presence of b in the ready set of $a.\,1 + b.\,0$. □

Intuitively, the action b is *unusable* for r after having performed the unsuccessful trace c; this is because performing b leads to a client, $\mathsf{0}$, which is *unusable*, in the sense that it can never be satisfied by any server. When comparing ready sets after unsuccessful traces in Definition 4 we should ignore occurrences of *unusable* actions.

Let $\mathcal{U}\mathsf{clt} = \{ r \mid p \ \mathsf{must}\ r, \text{ for some server } p \}$. The set $\mathcal{U}\mathsf{clt}$ contains the usable clients, those satisfied by at least one server. We also need to consider the residuals of a client r only after unsuccessful traces: for any process r and $s \in \mathsf{Act}^\star$ let

$$(r \ \mathsf{after}_{\not\checkmark}\ s) = \{ q \mid r \stackrel{s}{\Longrightarrow}_{\not\checkmark} q \}$$

Now the set of usable actions for a client after s can be defined as

$$\mathsf{ua}_{\mathsf{clt}}(r, s) = \{ a \in \mathsf{Act} \mid \bigoplus(r \ \mathsf{after}_{\not\checkmark}\ sa) \in \mathcal{U}\mathsf{clt} \} \tag{3}$$

Thus if $a \in \mathsf{ua}_{\mathsf{clt}}(r, s)$ we know that the set of clients $(r \ \mathsf{after}_{\not\checkmark}\ sa)$ is non-empty, and the client given by the choice among those clients is usable; that is, there is some server which satisfies it.

Example 3. We revisit Example 2. Although r can perform the sequence cb, b is not in $\mathsf{ua}_{\mathsf{clt}}(r, c)$ because $(r \ \mathsf{after}_{\not\checkmark}\ cb)$ is the singleton set containing $\mathsf{0}$, which is not in $\mathcal{U}\mathsf{clt}$. Instead we have $\mathsf{ua}_{\mathsf{clt}}(r, c) = \{ a \}$.

If we amend Definition 4 by replacing the set inclusion $A \subseteq B$ with the more relaxed condition $A \cap \mathsf{ua}_{\mathsf{clt}}(r_1, s) \subseteq B$, it follows that $r \preccurlyeq_{\mathsf{bad}} c.a.\,1$; thereby correctly reflecting the fact that $r \sqsubseteq_{\mathsf{clt}} c.a.\,1$. □

Example 4. In (3) above we must consider only the unsuccessful traces rather than all the traces. Consider the client $r = b.(\tau.(1 + a.\,0) + \tau.a.\tau.\,1)$. First note that $\overline{b}.\overline{a}.\,0 \ \mathsf{must}\ r$ while $\overline{b}.\overline{a}.\,0 \not\mathsf{must}\ b.\,0$ and therefore $r \not\sqsubseteq_{\mathsf{clt}} b.\,0$.

Now consider the consequences of using after rather than $\mathsf{after}_{\not\checkmark}$ in the definition (3) above. The amendment to the definition of $\preccurlyeq_{\mathsf{bad}}$ suggested in Example 3 would no longer be sound, as $r \preccurlyeq_{\mathsf{bad}} b.\,0$ would be true.

This is because $(r \ \mathsf{after}\ ba)$ is the set $\{ \mathsf{0}, \mathsf{1} \}$ and so $\bigoplus(r \ \mathsf{after}\ ba)$ is the client $\tau.\,\mathsf{0} + \tau.\,\mathsf{1}$, which is not in $\mathcal{U}\mathsf{clt}$. This leads to $\mathsf{ua}_{\mathsf{clt}}(r, b)$ being $\emptyset$, from which $r \preccurlyeq_{\mathsf{bad}} b.\,0$ would follow. The incorrect reasoning involves the unsuccessful acceptance set after the trace b. $\mathsf{Acc}_{\not\checkmark}(b.\,0, b) = \{ \emptyset \}$ and the unique ready set it contains, $\emptyset$, can be matched by $A \cap \emptyset$ for some $A \in \mathsf{Acc}_{\not\checkmark}(r, b)$, namely $A = \{ a \}$.

However with the correct definition (3) this reasoning no longer works as $\mathsf{ua}_{\mathsf{clt}}(r, b) = \{ a \}$. □

Unfortunately the amendment to Definition 4 suggested in Example 3 is still not sufficient to obtain a complete characterisation of the client preorder.

Example 5. Consider the clients $r_1 = a.(b.d.\,\mathsf{0} + b.\,\mathsf{1})$ and $r_2 = a.c.d.\,\mathsf{1}$. As r_1 is not usable $r_1 \sqsubseteq_{\mathsf{clt}} r_2$, although $r_1 \not\preccurlyeq_{\mathsf{bad}} r_2$, even when $\preccurlyeq_{\mathsf{bad}}$ is amended as suggested in Example 3. To see this first note $\{ d \} \in \mathsf{Acc}_{\not\checkmark}(r_2, ac)$, and $r_1 \Downarrow ac$, although r_1 can not actually perform the sequence of actions ac; $r_1 \Downarrow ac$ merely says that if r_1 can perform any prefix of the sequence ac to reach r' then r' must converge. Consequently $\mathsf{Acc}_{\not\checkmark}(r_1, ac)$ is empty and thus no ready set B can be found to match the ready set $\{ d \}$. □

To fix this problem we need to reconsider when ready sets are to be matched. In Definition 4 this matching is moderated by the predicate $\Downarrow s$; for example $a.(\tau^\infty + b.1) \preccurlyeq_{\mathsf{bad}} a.c.d.1$, where τ^∞ denotes some process which does not converge. This is because $a.(\tau^\infty + b.1) \Downarrow a$ is false and therefore the ready set $\{c\} \in \mathsf{Acc}_{\not\downarrow}(a.c.d.1, a)$ does not have to be matched by $a.(\tau^\infty + b.1)$. However the client preorder is largely impervious to convergence/divergence. For example $1 \approx_{\mathsf{clt}} (1 + \tau^\infty)$.

It turns out that we have to moderate the matching of ready sets, not via the convergence predicate, but instead via *usability*.

For every $s \in \mathsf{Act}^\star$, the *client* usability after an unsuccessful trace s, denoted $\mathsf{usbl}_{\not\downarrow}\ s$, is defined by induction on s:

- $r\ \mathsf{usbl}_{\not\downarrow}\ \varepsilon$ if $r \in \mathcal{U}\mathsf{clt}$
- $r\ \mathsf{usbl}_{\not\downarrow}\ a.s$ if $r \in \mathcal{U}\mathsf{clt}$, and if $r \stackrel{a}{\Longrightarrow}_{\not\downarrow}$ then $\bigoplus(r\ \mathsf{after}_{\not\downarrow}\ a)\ \mathsf{usbl}_{\not\downarrow}\ s$

It is extended to infinite traces $u \in \mathsf{Act}^\infty$ in the obvious manner. Intuitively $r\ \mathsf{usbl}_{\not\downarrow}\ s$ means that any state reachable from r by performing any subsequence of s is usable. Note that only unsuccessful traces have to be taken into the account.

One can show that if $r_1 \sqsubseteq_{\mathsf{clt}} r_2$ and $r_1\ \mathsf{usbl}_{\not\downarrow}\ s$ then $r_2\ \mathsf{usbl}_{\not\downarrow}\ s$. In fact this predicate describes precisely when we expect ready sets of clients to be compared.

Definition 5 (Semantic client-preorder). *In any LTS, let $r_1 \precsim_{\mathsf{clt}} r_2$ if (1) for every $s \in \mathsf{Act}^\star$ such that $r_1\ \mathsf{usbl}_{\not\downarrow}\ s$, (a) $r_2\ \mathsf{usbl}_{\not\downarrow}\ s$, and (b) for every $B \in \mathsf{Acc}_{\not\downarrow}(r_2, s)$ there exists some $A \in \mathsf{Acc}_{\not\downarrow}(r_1, s)$ such that*

$$A \cap \mathsf{ua}_{\mathsf{clt}}(r_1, s) \subseteq B$$

(2) for $w \in \mathsf{Act}^\star \cup \mathsf{Act}^\infty$ such that $r_1\ \mathsf{usbl}_{\not\downarrow}\ w$, $r_2 \stackrel{w}{\Longrightarrow}_{\not\downarrow}$ implies $r_1 \stackrel{w}{\Longrightarrow}_{\not\downarrow}$. □

Example 6. Let us revisit the clients r_1, r_2, in Example 5. The client $b.d.0 + b.1$ is not usable; that is $b.d.0 + b.1 \notin \mathcal{U}\mathsf{clt}$ because it cannot be satisfied by any server. Consequently $r_1\ \mathsf{usbl}_{\not\downarrow}\ ac$ does not hold, and therefore when checking whether $r_1 \precsim_{\mathsf{clt}} r_2$ holds the ready set $\{d\} \in \mathsf{Acc}_{\not\downarrow}(r_2, ac)$ does not have to be matched by r_1.

Indeed it is now straightforward to check that $r_1 \precsim_{\mathsf{clt}} r_2$; the only $s \in \mathsf{Act}^\star$ for which $\mathsf{Acc}_{\not\downarrow}(r_2, s)$ is non-empty and $r_1\ \mathsf{usbl}_{\not\downarrow}\ s$ is the empty sequence ε. □

In general, and in particular in LTSs which are not finite branching, the condition on the existence of infinite computations in (2) does not follow from the condition on finite computations.

Example 7. Consider the process q from Figure 3, where q_k denotes a process which performs a sequence of k a actions followed by 1. Let p be a similar process, but without the self loop. Then $p\ \mathsf{usbl}_{\not\downarrow}\ s$ and $q\ \mathsf{usbl}_{\not\downarrow}\ s$ for every s, and the pair (p, q) satisfies condition (1) of $\precsim_{\mathsf{clt}}$, and condition (2) on finite ws. However condition (2) on infinite ws is not satisfied: if u denotes the infinite sequence of as then $q \stackrel{u}{\Longrightarrow}_{\not\downarrow}$ but $p \stackrel{u}{\not\Longrightarrow}_{\not\downarrow}$.

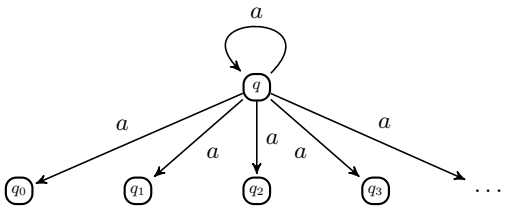


Fig. 3. Infinite traces

In fact $p \not\sqsubseteq_{\mathsf{clt}} q$. For consider the process $A \stackrel{\mathsf{def}}{=} \overline{a}.A$. When p is run as a test on A, or as a *client* using the *server* A, every computation is finite and successful; A must p. However when q is run as a test, there is the possibility of an infinite computation, the indefinite synchronisation on a, which is not successful; A $\not$must q. □

Theorem 2. In CCS, $r_1 \sqsubseteq_{\mathsf{clt}} r_2$ if and only if $r_1 \lesssim_{\mathsf{clt}} r_2$. □

The server-preorder $\sqsubseteq_{\mathsf{svr}}$ can be characterised behaviourally in manner dual to that of Definition 5, using the set of usable servers $\mathcal{U}\mathsf{svr} = \{ p \mid p \text{ must } r, \text{ for some client } r \}$, the usable actions $\mathsf{ua}_{\mathsf{svr}}(p,s) = \{ a \in \mathsf{Act} \mid \bigoplus(p \text{ after } sa) \in \mathcal{U}\mathsf{svr} \}$, and the *server* convergence predicate $p \Downarrow_{\mathsf{svr}} s$, defined as the conjunction of $p \Downarrow s$ and a server usability predicate p usbl s. This latter predicate is defined inductively in a manner similar to $\mathsf{usbl}\not\swarrow s$, but over all traces s, rather than simply the unsuccessful ones.

Definition 6 (Semantic server-preorder). *In any LTS, let $p \lesssim_{\mathsf{svr}} q$ if (1) for every $s \in \mathsf{Act}^\star$ such that $p \Downarrow_{\mathsf{svr}} s$, (a) $q \Downarrow_{\mathsf{svr}} s$, and (b) for every $B \in \mathsf{Acc}(q,s)$ there exists some $A \in \mathsf{Acc}(p,s)$ such that*

$$A \cap \mathsf{ua}_{\mathsf{svr}}(p,s) \subseteq B$$

(2) for every $w \in \mathsf{Act}^\star \cup \mathsf{Act}^\infty$ such that $p \Downarrow_{\mathsf{svr}} w$, $q \stackrel{w}{\Longrightarrow}$ implies $p \stackrel{u}{\Longrightarrow}$. □

Theorem 3. In CCS, $p \sqsubseteq_{\mathsf{svr}} q$ if and only if $p \lesssim_{\mathsf{svr}} q$. □

This can be seen to be a generalisation of Theorem 1, as the server usability predicate $\mathcal{U}\mathsf{svr}$ is degenerate; it holds for every process, since any process used as a server trivially satisfies the degenerate client 1.

Let us now consider the peer preorder. The following result is hopeful:

Proposition 4 In CCS, $p \sqsubseteq_{\mathsf{p2p}} q$ implies $p \sqsubseteq_{\mathsf{clt}} q$. □

Unfortunately, the peer preorder is *not* contained in the server preorder:

(S1a)	$\mu.x_{\not\checkmark} + \mu.y = \mu.(\tau.x_{\not\checkmark} + \tau.y)$	**(S3)**	$\mu.x + \tau.(\mu.y + z) = \tau.(\mu.x + \mu.y + z)$
(S1b)	$\tau.x \leq \tau.\tau.x$	**(S4)**	$\tau.x + \tau.y \leq x$
(S2)	$x_{\not\checkmark} + \tau.y = \tau.(x_{\not\checkmark} + y) + \tau.y$	**(S5)**	$\Omega \leq x$

Fig. 4. Standard inequations

Example 8. It is easy to see that $a.\,0 \sqsubseteq_{\approx_{\mathsf{p2p}}} b.\,0$. This is true because $a.\,0$ can never be satisfied, for it offers no $\checkmark$ at all. However, $a.\,0 \not\sqsubseteq_{\approx_{\mathsf{svr}}} b.\,0$, as the client $\overline{a}.\,1$ is satisfied by $a.\,0$, whereas $b.\,0 \not\mathsf{must}\ \overline{a}.\,1$. □

Intuitively, the reason why $\sqsubseteq_{\approx_{\mathsf{p2p}}} \not\subseteq \sqsubseteq_{\approx_{\mathsf{svr}}}$ is that the server preorder does not take into account the requirement that servers should now act as peers; they should also be satisfied by their interactions with clients. To take this into account we introduce the usability of peers and amend the definition of $\precsim_{\mathsf{svr}}$ accordingly. In principle we should introduce the set of usable peers, $\mathcal{U}\mathsf{p2p} = \{\, p \mid p\ \mathsf{must}^{\mathsf{p2p}}\ r \text{ for some peer } r \,\}$. However, since $\mathcal{U}\mathsf{p2p}$ turns out to coincide with $\mathcal{U}\mathsf{clt}$, instead we define the peer convergence predicate by using the usability predicate of *clients*. For every $w \in \mathsf{Act}^\star \cup \mathsf{Act}^\infty$, let $p \Downarrow_{\mathsf{p2p}} w$ whenever $p \Downarrow w$ and $p\ \mathsf{usbl}_{\not\checkmark}\ w$.

Definition 7. *Let $p \precsim_{u\mathsf{svr}} q$ whenever (1) for every $s \in \mathsf{Act}^\star$, if $p \Downarrow_{\mathsf{p2p}} s$ then (a) $q \Downarrow_{\mathsf{p2p}} s$, and (b) for every $B \in \mathsf{Acc}(q,s)$ there exists some $A \in \mathsf{Acc}(p,s)$ such that*

$$A \cap \mathsf{ua}_{\mathsf{clt}}(p,s) \subseteq B$$

(2) for every $w \in \mathsf{Act}^\star \cup \mathsf{Act}^\infty$, if $p \Downarrow_{\mathsf{p2p}} w$, and $q \overset{w}{\Longrightarrow}$, then $p \overset{w}{\Longrightarrow}$. □

Definition 8 (Semantic peer-preorder). *Let $p \precsim_{\mathsf{p2p}} q$ if $p \precsim_{\mathsf{clt}} q$ and $p \precsim_{u\mathsf{svr}} q$.*

Note that the definition of $p \precsim_{\mathsf{p2p}} q$ is **not** simply the conjunction of the client and server preorders from Definition 5 and Definition 6. It is essential that the usable set of peers $\mathcal{U}\mathsf{p2p}$ be employed.

Theorem 5. In CCS, $p \sqsubseteq_{\approx_{\mathsf{p2p}}} q$ if and only if $p \precsim_{\mathsf{p2p}} q$. □

4 Equational Characterisation

We use $\mathsf{CCS}^{\mathsf{f}}$ to denote the finite sub-language of CCS; this consists of all finite words constructed from the operators $\mathsf{0}$, $\mathsf{1}$, $+$, $\mu._{-}$ for each $\mu \in \mathsf{Act}_\tau$, together with the special operator Ω; this last denotes the term τ^∞ from CCS and its inclusion enables us to consider the algebraic properties of divergent processes. Our intention is to use equations, or more generally inequations, to characterise the three behavioural preorders $p \sqsubseteq_\star q$ over this finite algebra, where $\star$ ranges over svr, clt and p2p. Minor variations on standard equations, [Mil89], can be used for the other operators, such as parallel and hiding. For a given set of

(**Za**)	$\tau.\mathsf{0} \leq \Omega$	(**Zb**)	$\mu.\mathsf{0} \leq \mathsf{0}$
(**CLT1a**)	$x \leq \mathsf{1}$	(**P2P1**)	$\mathsf{0} \leq \mathsf{1}$
(**CLT1b**)	$\mathsf{1} \leq x + \mathsf{1}$	(**P2P2**)	$\mu.(\mathsf{1} + x) \leq \mathsf{1} + \mu.x$
(**CLT1c**)	$\mathsf{0} \leq \mu.\mathsf{1}$	(**P2P3**)	$\mu.(\mathsf{1} + x) + \mu.(\mathsf{1} + y) \leq \mu.(\mathsf{1} + \tau.x + \tau.y)$

Fig. 5. Client and peer inequations

inequations E we will use $p \sqsubseteq_E q$ to denote the fact that the inequation $p \leq q$ can be derived from E using standard equational reasoning, while $p =_E q$ means that both $p \sqsubseteq_E q$ and $q \sqsubseteq_E p$ can be derived.

There are two immediate obstacles. The first is that these preorders are not pre-congruences for the language $\mathsf{CCS}^{\mathrm{f}}$; specifically they are not preserved by the choice operator $+$.

Example 9. Using the behavioural characterisation in Definition 8 it is easy to check that $\mathsf{0} \mathrel{\sqsubseteq\!\!\!\approx}_{\mathsf{p2p}} b.\mathsf{0}$; in fact this is trivial because $\mathsf{0} \notin \mathcal{U}\mathsf{p2p}$. However $a.\mathsf{1} + \mathsf{0} \not\mathrel{\sqsubseteq\!\!\!\approx}_{\mathsf{p2p}} a.\mathsf{1} + b.\mathsf{1}$ because $\overline{a}.\mathsf{1} + \overline{b}.\mathsf{0}\ \ \mathsf{must}^{\mathsf{p2p}}\ \ a.\mathsf{1} + \mathsf{0}$ while $\overline{a}.\mathsf{1} + \overline{b}.\mathsf{0}\ \not\mathsf{must}^{\mathsf{p2p}}\ a.\mathsf{1} + b.\mathsf{0}$; the latter follows because of the possible communication on b.

The same counter-example also shows that the other preorders are also not preserved. □

So in order to discuss equational reasoning we focus on the largest $\mathsf{CCS}^{\mathrm{f}}$ pre-congruence contained in $\mathrel{\sqsubseteq\!\!\!\approx}_{\star}$ which we denote by $\mathrel{\sqsubseteq\!\!\!\approx}^{\mathrm{c}}_{\star}$; by definition this is preserved by all the operators. But it is convenient to have alternative more amenable characterisations. To this end we let $p \mathrel{\sqsubseteq\!\!\!\approx}^{+}_{\star} q$ to mean that $a.\mathsf{1} + p \mathrel{\sqsubseteq\!\!\!\approx}_{\star} a.\mathsf{1} + q$ for some fresh action name a.

Proposition 6 In CCS, $p \mathrel{\sqsubseteq\!\!\!\approx}^{\mathrm{c}}_{\star} q$ if and only if $p \mathrel{\sqsubseteq\!\!\!\approx}^{+}_{\star} q$. □

Note that this is similar to the characterisation of observation-congruence in Section 7.2 of [Mil89]; the same technique is also used in [NH84].

The second obstacle is that the behavioural preorders are very sensitive to the ability of processes to immediately report success, with the result that many of the expected equations are not in general valid. For example the innocuous

$$a.\tau.x = a.x,$$

valid in the theories of [Mil89, NH84], is not in general satisfied by two of our behavioural theories. For example $a.\mathsf{1} \not\mathrel{\sqsubseteq\!\!\!\approx}^{+}_{\mathsf{p2p}} a.\tau.\mathsf{1}$ because of the peer $\overline{a}.(\mathsf{1} + \Omega)$.

In order to have a more elegant presentation of the axioms we will use two sorts of variables, the standard $x, y, \ldots$ which may be instantiated with any process from $\mathsf{CCS}^{\mathrm{f}}$, and $x\!\!\not\!\!\checkmark, y\!\!\not\!\!\checkmark, \ldots$ which may only be instantiated by a process p

satisfying $p \not\xrightarrow{}$; in $\mathsf{CCS}^{\mathsf{f}}$ such processes p in fact have a simple syntactic characterisation. With this convention in mind consider the five *standard* inequations given in Figure 4, which are satisfied by all three behavioural preorders $\sqsubseteq^{+}_{\star}$. We also assume the standard equations for $(\mathsf{CCS}^{\mathsf{f}}, +, \mathsf{0})$ being a commutative monoid. Let **SVR** denote the set of inequations obtained by adding

$$(\mathbf{SVR1}) \qquad \mathsf{1} = \mathsf{0}$$

Intuitively $\mathsf{1}$ has no significance for server behaviour; this extra equation captures this intuition and is sufficient to characterise the server preorder:

Theorem 7 [Soundness and completeness for server-testing] In $\mathsf{CCS}^{\mathsf{f}}$, $p \sqsubseteq^{\mathsf{c}}_{\approx_{\mathsf{svr}}} q$ if and only $p \sqsubseteq_{\mathbf{SVR}} q$. □

In order to characterise the client and peer preorders we need to replace the equation **SVR1** with inequations which capture the significance of the operators $\mathsf{1}$ and $\mathsf{0}$ for clients and peers respectively. A sufficient set of inequations for clients is also given in Figure 4. Thus the client preorder has both a least element Ω from (**S5**), which by (**Za**) is also equivalent to $\tau.\mathsf{0}$, and a greatest element $\mathsf{1}$ from (**CLT1a**). Let **CLT** denote the resulting set of inequations.

Theorem 8 [Soundness and Completeness for client-testing] In $\mathsf{CCS}^{\mathsf{f}}$, $p \sqsubseteq^{\mathsf{c}}_{\approx_{\mathsf{clt}}} q$ if and only $p \sqsubseteq_{\mathbf{CLT}} q$. □

Both the inequations (**Za**) and (**Zb**) remain valid for the peer preorder, but none of the unit inequations (**CLT1a**) - (**CLT1c**) are. They need to be replaced by unit inequations appropriate to peers. Let **P2P** denote the set obtained by replacing them with (**P2P1**) - (**P2P3**).

Theorem 9 [Soundness and Completeness for peer-testing] In $\mathsf{CCS}^{\mathsf{f}}_{\mathsf{w}\tau}$, $p \sqsubseteq^{\mathsf{c}}_{\approx_{\mathsf{p2p}}} q$ if and only $p \sqsubseteq_{\mathbf{P2P}} q$. □

5 Conclusions

Much of the recent work on behavioural preorders for processes has been carried out using formalisms for contracts for web-services, proposed first in [CCLP06]. Spurred on by the recasting of the standard must preorder from [NH84] as a server-preorder between contracts, these ideas have been developed further in [LP07, CGP09, Bd10, Pad10].

In these publications the standard refinements are referred to as *subcontracts* or *sub-server* relations and [LP07, CGP09, Pad10, Bd10] contain a range of alternative characterisations. For example in [LP07, CGP09] the characterisations are coinductive and essentially rely on traces and ready sets; in [Bd10] the characterisation is coinductive and syntax-oriented.

To the best of our knowledge, the first paper to use a preorder for clients is [Bd10]. But their setting is much more restricted; they use so-called *session*

behaviours which correspond to a much smaller class of processes than our language CCS. As there are fewer contexts, their sub-server preorder differs from our server preorder: $a_1.1 \preceq_s a_1.1 + a_2.1$, whereas $a_1.1 \not\sqsubseteq_{\mathsf{svr}} a_1.1 + a_2.1$.

The refinements in the papers mentioned above depend on a *compliance* relation, rather than must testing; this is also why in [Bd10] the peer preorder $\preceq$: coincides with the intersection of the client and the server preorders; this is not the case for the must preorders (Example 8 can be tailored to the setting of session behaviours). Moreover, in a general infinite branching and non-deterministic LTS the refinements in the above papers differ from the preorder $\sqsubseteq_{\mathsf{svr}}$. The subcontract relation of [LP07] turns out to be not comparable with $\sqsubseteq_{\mathsf{svr}}$, whereas the strong subcontract $\sqsubseteq$ of [Pad10] is strictly contained in $\sqsubseteq_{\mathsf{svr}}$, as the LTS there is convergent and finite branching. The comparison of $\sqsubseteq_{\mathsf{svr}}$ with the refinement preorder of [CGP09] is complicated by their use of a non-standard LTS.

In [BMPR09] a symmetric refinement due to the compliance, $\sqsubseteq^{\mathsf{ds}}$, is studied; it differs from our peer preorder ($\sqsubseteq_{\mathsf{p2p}} \not\subseteq \sqsubseteq^{\mathsf{ds}}$), and its characterisation does not mention usability. This is because of the restrictions of the LTS in [BMPR09]. In more general settings the usability of contracts/services is crucial; [Pad11] talks of *viability*, while [MSV10] talks of *controllability*.

Also subcontracts/subtyping for peers inspired by the should/fair-testing of [RV07] have been proposed in [BZ09, BMPR09, Pad11]. In [BZ09] the fair-testing preorder is used as proof method for relating contracts, but no characterisation of their refinement preorder is given. A sound but incomplete characterisation is given in [BMPR09]. The focus of [Pad11] is on multi-party *session types* which, roughly speaking, cannot express all the behaviours of our language CCS. In view of the restricted form of session types, they can give a syntax-oriented characterisation of their subtyping relation, $\leqslant$; this is in general incomparable with our $\sqsubseteq_{\mathsf{p2p}}$.

Future work: The most obvious open question about our two new refinement preorders $\sqsubseteq_{\mathsf{clt}}$ and $\sqsubseteq_{\mathsf{p2p}}$ is the development of algorithms for finite-state systems. The ability to check efficiently whether a process is *usable* will play an important role.

Another interesting question would be to characterise in some equational manner the refinement preorders $\sqsubseteq_{\mathsf{clt}}$, $\sqsubseteq_{\mathsf{p2p}}$ themselves rather than their associated pre-congruences $\sqsubseteq^{+}_{\mathsf{clt}}$ and $\sqsubseteq^{+}_{\mathsf{p2p}}$. In the resulting equational theory we would have to restrict in some way the form of reasoning allowed under the external choice operator $- + -$, but the extra inequations needed in such a proof system might be simpler.

We have confined our attention to refinement preorders based on must testing. But one can also define client and peer preorders based on the standard *may testing* of [NH84]. We believe that these refinement preorders can be completely characterised using a modified notion of *trace*, which takes into account the *usability* of residuals. Other variations on client and peer preorders are worth investigating: a "synchronous" formulation of $\sqsubseteq_{\mathsf{p2p}}$ where a computation is successful only if the peers report success *at the same time*; the client preorders for fair settings [Pad11, BZ09], or the ones based on the compliance [Pad10].

Acknowledgements. The first author would like to acknowledge Vasileios Koutavas, for his help in unravelling the client preorder.

References

[Bd10] Barbanera, F., de'Liguoro, U.: Two notions of sub-behaviour for session-based client/server systems. In: Kutsia, T., Schreiner, W., Fernández, M. (eds.) PPDP, pp. 155–164. ACM (2010)

[Ber13] Bernardi, G.: Behavioural Equivalences for Web Services. PhD thesis, Trinity College Dublin (2013), `https://www.scss.tcd.ie/~bernargi`

[BMPR09] Bugliesi, M., Macedonio, D., Pino, L., Rossi, S.: Compliance preorders for web services. In: Laneve, C., Su, J. (eds.) WS-FM 2009. LNCS, vol. 6194, pp. 76–91. Springer, Heidelberg (2010)

[BZ09] Bravetti, M., Zavattaro, G.: Contract-based discovery and composition of web services. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 261–295. Springer, Heidelberg (2009)

[CCLP06] Carpineti, S., Castagna, G., Laneve, C., Padovani, L.: A formal account of contracts for web services. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 148–162. Springer, Heidelberg (2006)

[CGP09] Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. ACM Trans. Program. Lang. Syst. 31(5), 1–61 (2009); Supersedes the article in POPL 2008 (2008)

[DH87] De Nicola, R., Hennessy, M.: CCS without tau's. In: Ehrig, H., Kowalski, R.A., Levi, G., Montanari, U. (eds.) GI 1973. LNCS, vol. 249, pp. 138–152. Springer, Heidelberg (1973)

[Hen88] Hennessy, M.: Algebraic Theory of Processes. The MIT Press, Cambridge (1988)

[Hoa85] Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall (1985)

[LP07] Laneve, C., Padovani, L.: The must preorder revisited. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 212–225. Springer, Heidelberg (2007)

[Mil89] Milner, R.: Communication and Concurrency. Prentice-Hall (1989)

[MSV10] Mooij, A.J., Stahl, C., Voorhoeve, M.: Relating fair testing and accordance for service replaceability. J. Log. Algebr. Program. 79(3-5), 233–244 (2010)

[NH84] De Nicola, R., Hennessy, M.C.B.: Testing equivalences for processes. Theoretical Computer Science 34(1-2), 83–133 (1984)

[Pad10] Padovani, L.: Contract-based discovery of web services modulo simple orchestrators. Theor. Comput. Sci. 411(37), 3328–3347 (2010)

[Pad11] 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)

[RV07] Rensink, A., Vogler, W.: Fair testing. Information and Computation 205(2), 125–198 (2007)

Hennessy-Milner Logic with Greatest Fixed Points as a Complete Behavioural Specification Theory

Nikola Beneš[1,⋆], Benoît Delahaye[2], Uli Fahrenberg[2], Jan Křetínský[1,3,⋆⋆], and Axel Legay[2]

[1] Masaryk University, Brno, Czech Republic
[2] Irisa / INRIA Rennes, France
[3] Technische Universität München, Germany

Abstract. There are two fundamentally different approaches to specifying and verifying properties of systems. The *logical* approach makes use of specifications given as formulae of temporal or modal logics and relies on efficient model checking algorithms; the *behavioural* approach exploits various equivalence or refinement checking methods, provided the specifications are given in the same formalism as implementations.

In this paper we provide translations between the logical formalism of Hennessy-Milner logic with greatest fixed points and the behavioural formalism of disjunctive modal transition systems. We also introduce a new operation of quotient for the above equivalent formalisms, which is adjoint to structural composition and allows synthesis of missing specifications from partial implementations. This is a substantial generalisation of the quotient for deterministic modal transition systems defined in earlier papers.

1 Introduction

There are two fundamentally different approaches to specifying and verifying properties of systems. Firstly, the *logical* approach makes use of specifications given as formulae of temporal or modal logics and relies on efficient model checking algorithms. Secondly, the *behavioural* approach exploits various equivalence or refinement checking methods, provided the specifications are given in the same formalism as implementations.

In this paper, we discuss different formalisms and their relationship. As an example, let us consider labelled transition systems and the property that *"at all time points after executing* request*, no* idle *nor further requests but only* work *is allowed until* grant *is executed"*. The property can be written in *e.g.* CTL [14] as

$$\mathrm{AG}(\mathsf{request} \Rightarrow \mathrm{AX}(\mathsf{work}\ \mathrm{AW}\ \mathsf{grant}))$$

⋆ The author has been supported by the Czech Science Foundation grant No. GAP202/11/0312.

⋆⋆ The author is partially supported by the Czech Science Foundation, project No. P202/10/1469.

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 76–90, 2013.

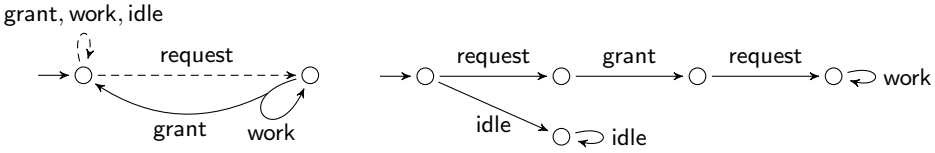


Fig. 1. DMTS specification corresponding to AG(request ⇒ AX(work AW grant)), and its implementation

or as a recursive system of equations in Hennessy-Milner logic [29] as

$$X = [\mathsf{grant}, \mathsf{idle}, \mathsf{work}]X \wedge [\mathsf{request}]Y$$
$$Y = (\langle \mathsf{work} \rangle Y \vee \langle \mathsf{grant} \rangle X) \wedge [\mathsf{idle}, \mathsf{request}]\mathbf{ff}$$

where the solution is given by the greatest fixed point.

As formulae of modal logics can be difficult to read, some people prefer automata-based behavioural specifications to logical ones. One such behavioural specification formalism is the one of disjunctive modal transition systems (DMTS) [26]. Fig. 1 (left) displays a specification of our example property as a DMTS. Here the dashed arrows indicate that the transitions *may or may not* be present, while branching of the solid arrow indicates that at least one of the branches *must* be present. An example of a labelled transition system that *satisfies* our logical specifications and *implements* the behavioural one is also given in Fig. 1.

The alternative between logical and behavioural specifications is not only a question of preference. Logical specification formalisms put a powerful logical language at the disposal of the user, and the logical approach to model checking [14,34] has seen a lot of success and tool implementations. Automata-based specifications [12,27], on the other hand, have a focus on *compositional* and *incremental* design in which logical specifications are somewhat lacking, with the trade-off of generally being less expressive than logics.

To be more precise, automata-based specifications are, by design, compositional in the sense that they support structural *composition* of specifications and, in most cases, its adjoint, *quotient*. This is useful, even necessary, in practical verification, as it means that (1) it is possible to infer properties of a system from the specifications of its components, and (2) the problem of correctness for a system can be decomposed into verification problems for its components. We refer to [28] for a detailed account on composition and decomposition.

It is thus desirable to be able to translate specifications from the logical realm into behavioural formalisms, and *vice versa* from behavioural formalisms to logic-based specifications. This is, then, the first contribution of this paper: we show that Hennessy-Milner logic with greatest fixed points (νHML) and DMTS (with several initial states) are equally expressive, and we provide translations forth and back. For doing this, we introduce an auxiliary intermediate formalism NAA (a nondeterministic extension of acceptance automata [22,35]) which is equivalent in expressiveness to both νHML and DMTS.

We also discuss other desirable features of specification formalisms, namely structural composition and quotient. As an example, consider a specification S

of the final system to be constructed and T either an already implemented component or a specification of a service to be used. The task is to construct the most general specification of the rest of the system to be implemented, in such a way that when composed with any implementation of T, it conforms with the specification S. This specification is exactly the quotient S/T.

Contribution. Firstly, we show that the formalisms of νHML, NAA and DMTS have the same expressive power, and provide the respective translations. As a result, the established connection allows for a graphical representation of νHML as DMTS. This extends the graphical representability of HML without fixed points as modal transition systems [10,27]. In some sense this is optimal, as due to the alternation of least and greatest fixed points, there seems to be no hope that the whole μ-calculus could be drawn in a similarly simple way.

Secondly, we show that there are natural operations of conjunction and disjunction for NAA which mimic the ones of νHML. As we work with multiple initial states, disjunction is readily defined, and conjunction extends the one for DMTS [6]. Thirdly, we introduce structural composition on NAA. For simplicity we assume CSP-style synchronisation of labels, but the construction can easily be generalised to other types of label synchronisation.

Finally, we provide a solution to the open problem of the general quotient. We extend the quotient constructions for deterministic modal transition systems (MTS) and acceptance automata [35] to define the quotient for the full class of (possibly nondeterministic) NAA. We also provide a more efficient procedure for (possibly nondeterministic) MTS. These constructions are the technically most demanding parts of the paper.

With the operations of structural composition and quotient, NAA, and hence also DMTS and νHML, are fully compositional behavioural specification theories and form a *commutative residuated lattice* [21,39] up to equivalence. This makes a rich algebraic theory available for compositional reasoning about specifications. Most of the constructions we introduce are implemented in a prototype tool [8]. Due to space constraints, some of the proofs had to be omitted from the paper and can be found in [3].

Related Work. Hennessy-Milner logic with recursion [29] is a popular logical specification formalism which has the same expressive power as μ-calculus [25]. It is obtained from Hennessy-Milner logic (HML) [23] by introducing variables and greatest and least fixed points. Hennessy-Milner logic with *greatest* fixed points (νHML) is equivalent to ν-calculus, *i.e.* μ-calculus with greatest fixed points only.

DMTS have been proposed as solutions to algebraic process equations in [26] and further investigated also as a specification formalism [6, 28]. The DMTS formalism is a member of the modal transition systems (MTS) family and as such has also received attention recently. The MTS formalisms have proven to be useful in practice. Industrial applications started as early as [11] where MTS have been used for an air-traffic system at Heathrow airport. Besides, MTS classes are advocated as an appropriate base for interface theories in [36] and

for product line theories in [31]. Further, an MTS based software engineering methodology for design via merging partial descriptions of behaviour has been established in [38] and methods for supervisory control of MTS shown in [15]. Tool support is quite extensive, *e.g.* [2, 6, 9, 16].

Over the years, many extensions of MTS have been proposed. While MTS can only specify whether or not a particular transition is required, some extensions equip MTS with more general abilities to describe what *combinations* of transitions are possible. These include DMTS [26], 1-MTS [17] allowing to express exclusive disjunction, OTS [4] capable of expressing positive Boolean combinations, and Boolean MTS [5] covering all Boolean combinations. The last one is closely related to our NAA, the acceptance automata of [22,35], as well as hybrid modal logic [7, 33].

Larsen has shown in [27] that any finite MTS is equivalent to a HML formula (without recursion or fixed points), the *characteristic formula* of the given MTS. Conversely, Boudol and Larsen show in [10] that any consistent and *prime* HML formula is equivalent to a MTS. Here we extend these results to νHML formulae, and show that any such formula is equivalent to a DMTS, solving a problem left open in [26]. Hence νHML supports full compositionality and decomposition in the sense of [28]. This finishes some of the work started in [10, 27, 28].

Quotients are related to *decomposition* of processes and properties, an issue which has received considerable attention through the years. In [26], a solution to bisimulation $C(X) \sim P$ for a given process P and context C is provided (as a DMTS). This solves the quotienting problem P/C for the special case where both P and C are processes. This is extended in [30] to the setting where the context C can have several holes and $C(X_1, \ldots, X_n)$ must satisfy a property Q of νHML. However, C remains to be a process context, not a specification context. Our *specification* context allows for arbitrary specifications, representing infinite sets of processes and process equations. Another extension uses infinite conjunctions [19], but similarly to the other approaches, generates partial specifications from an overall specification and a given set of processes. This is subsumed by a general quotient.

Quotient operators, or *guarantee* or *multiplicative implication* as they are called there, are also well-known from various logical formalisms. Indeed, the algebraic properties of our parallel composition $\|$ and quotient / resemble closely those of multiplicative conjunction & and implication $\multimap$ in *linear logic* [20], and of spatial conjunction and implication in *spatial logic* [13] and *separation logic* [32, 37]. For these and other logics, proof systems have been developed which allow one to reason about expressions containing these operators.

In spatial and separation logic, & and $\multimap$ (or the operators corresponding to these linear-logic symbols) are first-class operators on par with the other logical operators, and their semantics are defined as certain sets of processes. In contrast, for NAA and hence, via the translations, also for νHML, $\|$ and / are *derived* operators, and we provide constructions to reduce any expression which contains them, to one which does not. This is important from the perspective of reuse of components and useful in industrial applications.

2 Specification Formalisms

In this section, we define the specification formalisms νHML, DMTS and NAA and show that they are equivalent.

For the rest of the paper, we fix a finite alphabet Σ. In each of the formalisms, the semantics of a specification is a set of implementations, in our case always a set of *labelled transition systems* (LTS) over Σ, *i.e.* structures $(S, s^0, \longrightarrow)$ consisting of a set S of *states*, an initial state $s^0 \in S$, and a *transition relation* $\longrightarrow \subseteq S \times \Sigma \times S$. We assume that the transition relation of LTS is always *image-finite*, i.e. that for every $a \in \Sigma$ and $s \in S$ the set $\{s' \in S \mid s \xrightarrow{a} s'\}$ is finite.

2.1 Hennessy-Milner Logic with Greatest Fixed Points

We recap the syntax and semantics of HML with variables developed in [29]. A *HML formula* ϕ over a set X of variables is given by the abstract syntax $\phi ::= \mathbf{tt} \mid \mathbf{ff} \mid x \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle a \rangle \phi \mid [a]\phi$, where x ranges over X and a over Σ. The set of such formulae is denoted $\mathcal{H}(X)$. Notice that instead of including fixed point operators in the logic, we choose to use declarations with a greatest fixed point semantics, as explained below.

A *declaration* is a mapping $\Delta : X \to \mathcal{H}(X)$. We shall give a greatest fixed point semantics to declarations. Let $(S, s^0, \longrightarrow)$ be an LTS, then an *assignment* is a mapping $\sigma : X \to 2^S$. The set of assignments forms a complete lattice with $\sigma_1 \sqsubseteq \sigma_2$ iff $\sigma_1(x) \subseteq \sigma_2(x)$ for all $x \in X$ and $\left(\bigsqcup_{i \in I} \sigma_i\right)(x) = \bigcup_{i \in I} \sigma_i(x)$.

The semantics of a formula is a subset of S, given relative to an assignment σ, defined as follows: $(\!|\mathbf{tt}|\!)\sigma = S$, $(\!|\mathbf{ff}|\!)\sigma = \emptyset$, $(\!|x|\!)\sigma = \sigma(x)$, $(\!|\phi \wedge \psi|\!)\sigma = (\!|\phi|\!)\sigma \cap (\!|\psi|\!)\sigma$, $(\!|\phi \vee \psi|\!)\sigma = (\!|\phi|\!)\sigma \cup (\!|\psi|\!)\sigma$, $(\!|\langle a \rangle \phi|\!)\sigma = \{s \in S \mid \exists s \xrightarrow{a} s' : s' \in (\!|\phi|\!)\sigma\}$, and $(\!|[a]\phi|\!)\sigma = \{s \in S \mid \forall s \xrightarrow{a} s' : s' \in (\!|\phi|\!)\sigma\}$. The semantics of a declaration Δ is then the assignment defined by $(\!|\Delta|\!) = \bigsqcup\{\sigma : X \to 2^S \mid \forall x \in X : \sigma(x) \subseteq (\!|\Delta(x)|\!)\sigma\}$: the greatest (pre)fixed point of Δ.

An *initialised* HML declaration, or *νHML formula*, is a structure (X, X^0, Δ), with $X^0 \subseteq X$ finite sets of variables and $\Delta : X \to \mathcal{H}(X)$ a declaration. We say that an LTS $(S, s^0, \longrightarrow)$ *implements* (or models) the formula, and write $S \models \Delta$, if it holds that there is $x^0 \in X^0$ such that $s^0 \in (\!|\Delta|\!)(x^0)$. We write $[\![\Delta]\!]$ for the set of implementations (models) of a νHML formula Δ.

2.2 Disjunctive Modal Transition Systems

A DMTS is essentially a labelled transition system (LTS) with two types of transitions, *may* transitions which indicate that implementations are permitted to implement the specified behaviour, and *must* transitions which proclaim that any implementation is required to implement the specified behaviour. Additionally, *must* transitions may be *disjunctive*, in the sense that they can require that *at least one* out of a number of specified behaviours must be implemented. We now recall the syntax and semantics of DMTS as introduced in [26]. We modify the syntax slightly to permit multiple initial states and, in the spirit of later work [6, 18], ensure that all required behaviour is also allowed:

A *disjunctive modal transition system* (DMTS) over the alphabet Σ is a structure $(S, S^0, \dashrightarrow, \longrightarrow)$ consisting of a set of *states* S, a finite subset $S^0 \subseteq S$ of *initial states*, a *may*-transition relation $\dashrightarrow \subseteq S \times \Sigma \times S$, and a *disjunctive must*-transition relation $\longrightarrow \subseteq S \times 2^{\Sigma \times S}$. It is assumed that for all $(s, N) \in \longrightarrow$ and all $(a, t) \in N$, $(s, a, t) \in \dashrightarrow$. We usually write $s \overset{a}{\dashrightarrow} t$ instead of $(s, a, t) \in \dashrightarrow$ and $s \longrightarrow N$ instead of $(s, N) \in \longrightarrow$. We also assume that the may transition relation is image-finite. Note that the two assumptions imply that $\longrightarrow \subseteq S \times 2^{\Sigma \times S}_{\mathrm{Fin}}$ where 2^X_{Fin} denotes the set of all finite subsets of X.

A DMTS $(S, S^0, \dashrightarrow, \longrightarrow)$ is an *implementation* if $S^0 = \{s^0\}$ is a singleton and $\longrightarrow = \{(s, \{(a, t)\} \mid s \overset{a}{\dashrightarrow} t\}$, hence if N is a singleton for each $s \longrightarrow N$ and there are no superfluous may-transitions. Thus DMTS implementations are precisely LTS.

We proceed to define the semantics of DMTS. First, a relation $R \subseteq S_1 \times S_2$ is a *modal refinement* between DMTS $(S_1, S_1^0, \dashrightarrow_1, \longrightarrow_1)$ and $(S_2, S_2^0, \dashrightarrow_2, \longrightarrow_2)$ if it holds for all $(s_1, s_2) \in R$ that

- for all $s_1 \overset{a}{\dashrightarrow} t_1$ there is $s_2 \overset{a}{\dashrightarrow} t_2$ for some $t_2 \in S_2$ with $(t_1, t_2) \in R$, and
- for all $s_2 \longrightarrow N_2$ there is $s_1 \longrightarrow N_1$ such that for each $(a, t_1) \in N_1$ there is $(a, t_2) \in N_2$ with $(t_1, t_2) \in R$.

Such a modal refinement is *initialised* if it is the case that, for each $s_1^0 \in S_1^0$, there is $s_2^0 \in S_2^0$ for which $(s_1^0, s_2^0) \in R$. In that case, we say that S_1 refines S_2 and write $S_1 \leq_{\mathrm{m}} S_2$. We write $S_1 \equiv_{\mathrm{m}} S_2$ if $S_1 \leq_{\mathrm{m}} S_2$ and $S_2 \leq_{\mathrm{m}} S_1$.

We say that an LTS I *implements* a DMTS S if $I \leq_{\mathrm{m}} S$ and write $[\![S]\!]$ for the set of implementations of S. Notice that the notions of implementation and modal refinement agree, capturing the essence of DMTS as a *specification theory*: A DMTS may be *gradually* refined, until an LTS, in which all behaviour is fully specified, is obtained.

For DMTS S_1, S_2 we say that S_1 *thoroughly* refines S_2, and write $S_1 \leq_{\mathrm{t}} S_2$, if $[\![S_1]\!] \subseteq [\![S_2]\!]$. We write $S_1 \equiv_{\mathrm{t}} S_2$ if $S_1 \leq_{\mathrm{t}} S_2$ and $S_2 \leq_{\mathrm{t}} S_1$. By transitivity, $S_1 \leq_{\mathrm{m}} S_2$ implies $S_1 \leq_{\mathrm{t}} S_2$.

Example 1. Figs. 2 and 3 show examples of important basic properties expressed both as νHML formulae, NAA (see below) and DMTS. For DMTS, may transitions are drawn as dashed arrows and disjunctive must transitions as branching arrows. States with a short incoming arrow are initial (the DMTS in Fig. 3 has *two* initial states).

$$X = \langle a \rangle \mathrm{tt} \wedge [a]X \wedge [b]X$$

$(\{s_0\}, \{s_0\}, \mathrm{Tran})$
$\mathrm{Tran}(s_0) = \{\{(a, s_0)\}, \{(a, s_0), (b, s_0)\}\}$

Fig. 2. νHML formula, NAA and DMTS for the invariance property "*there is always an 'a' transition available*", with $\Sigma = \{a, b\}$

$$X = \langle b\rangle \mathbf{tt} \vee \big(\langle a\rangle \mathbf{tt} \wedge [a]X \wedge [b]X \wedge [c]X\big)$$

$$(\{s_0, s_1\}, \{s_0\}, \mathrm{Tran})$$
$$\mathrm{Tran}(s_0) = \big\{\{(b, s_1)\}, \{(b, s_1), (a, s_1)\}, \{(b, s_1), (c, s_1)\}, \{(b, s_1), (a, s_1), (c, s_1))\}, \{(a, s_0)\}, \{(a, s_0), (c, s_0)\}\big\}$$
$$\mathrm{Tran}(s_1) = 2^{\{s_1\}\times\{a,b,c\}}$$

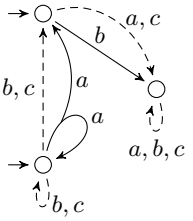
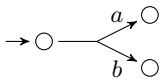


Fig. 3. νHML formula, NAA and DMTS for the ("weak until") property "*there is always an 'a' transition available, until a 'b' transition becomes enabled*", with $\Sigma = \{a, b, c\}$

Modal Transition Systems. An interesting subclass of DMTS are *modal transition systems* (MTS) [27]. A DMTS $(S, S^0, \dashrightarrow, \longrightarrow)$ is said to be a MTS if (1) $S^0 = \{s^0\}$ is a singleton, (2) for every $s \longrightarrow N$, the set N is a singleton. Hence, for each transition, we specify whether it must, may, or must not be present; no disjunctions can be expressed. It is easy to see that MTS are less expressive than DMTS, *i.e.* there are DMTS S for which no MTS S' exists so that $[\![S]\!] = [\![S']\!]$. One example is provided on the right. Here any implementation must have an a or a b transition from the initial state, but then any MTS which permits all such implementations will also allow implementations without any transition from the initial state.



2.3 NAA

We now define NAA, the nondeterministic extension to the formalism of acceptance automata [35]. We shall use this formalism to bridge the gap between νHML and DMTS. A *nondeterministic acceptance automaton* over the alphabet Σ is a structure (S, S^0, Tran) where S and S^0 are the states and initial states as previously, and $\mathrm{Tran} : S \to 2^{2^{\Sigma\times S}_{\mathrm{Fin}}}$ assigns admissible transition sets.

A NAA (S, S^0, Tran) is an *implementation* if $S^0 = \{s^0\}$ is a singleton and $\mathrm{Tran}(s) = \{M\}$ is a singleton for every $s \in S$; clearly, NAA implementations are precisely LTS. We also define the *inconsistent NAA* to be $\bot = (\emptyset, \emptyset, \emptyset)$ and the *universal NAA* by $\top = (\{s\}, \{s\}, 2^{2^{\Sigma\times\{s\}}})$.

A relation $R \subseteq S_1 \times S_2$ is a *modal refinement* between NAA $(S_1, S_1^0, \mathrm{Tran}_1)$, $(S_2, S_2^0, \mathrm{Tran}_2)$ if it holds for all $(s_1, s_2) \in R$ and all $M_1 \in \mathrm{Tran}_1(s_1)$ that there exists $M_2 \in \mathrm{Tran}_2(s_2)$ such that

- $\forall (a, t_1) \in M_1 : \exists (a, t_2) \in M_2 : (t_1, t_2) \in R$,
- $\forall (a, t_2) \in M_2 : \exists (a, t_1) \in M_1 : (t_1, t_2) \in R$.

We define and use the notions of initialised modal refinement, $\leq_{\mathrm{m}}$, $\equiv_{\mathrm{m}}$, implementation, $\leq_{\mathrm{t}}$, and $\equiv_{\mathrm{t}}$ the same way as for DMTS.

Proposition 2. *The class of NAA is preordered by modal refinement* $\leq_{\mathrm{m}}$*, with bottom element* $\bot$ *and top element* $\top$*.*

Note that as implementations of all our three formalisms νHML, DMTS and NAA are LTS, it makes sense to use thorough refinement $\leq_{\mathrm{t}}$ and equivalence $\equiv_{\mathrm{t}}$ *across* formalisms, so that we *e.g.* can write $S \leq_{\mathrm{t}} \Delta$ for a NAA S and a νHML formula Δ.

2.4 Equivalences

We proceed to show that νHML, DMTS and NAA are equally expressive:

Theorem 3. *For any set $\mathcal{S}$ of LTS, the following are equivalent:*

1. *There exists a νHML formula Δ with $[\![\Delta]\!] = \mathcal{S}$.*
2. *There exists a finite NAA S with $[\![S]\!] = \mathcal{S}$.*
3. *There exists a finite DMTS S with $[\![S]\!] = \mathcal{S}$.*

Furthermore, the latter two statements are equivalent even if we drop the finiteness constraints.

Note that we could drop the finiteness assumption about the set of variables of νHML formulae, while retaining the fact that $\Delta(x)$ is a finite HML formula. The result of Theorem 3 could then be extended with the statement that these possibly infinite νHML formulae are equivalent to general DMTS/NAA.

For a DMTS $S = (S, S^0, \dashrightarrow, \longrightarrow)$, let $\mathrm{Tran}(s) = \{M \subseteq \Sigma \times S \mid \exists N : s \longrightarrow N, N \subseteq M; \forall (a,t) \in M : s \overset{a}{\dashrightarrow} t\}$ and define the NAA $dn(S) = (S, S^0, \mathrm{Tran})$.

Conversely, for an NAA (S, S^0, Tran), define the DMTS $nd(S) = (T, T^0, \dashrightarrow, \longrightarrow)$ as follows:

- $T = \{M \in \mathrm{Tran}(s) \mid s \in S\}$, $T^0 = \{M \in \mathrm{Tran}(s^0) \mid s^0 \in S^0\}$,
- $\longrightarrow = \{(M, \{(a, M') \mid M' \in \mathrm{Tran}(s')\} \mid (a, s') \in M\}$,
- $\dashrightarrow = \{(t, a, t') \mid t \in T, \exists (t, N) \in \longrightarrow : (a, t') \in N\}$.

Note that both nd and dn preserve finiteness. Both translation are exponential in their respective arguments.

Lemma 4. *For every DMTS S, $S \equiv_{\mathrm{t}} dn(S)$. For every NAA S, $S \equiv_{\mathrm{t}} nd(S)$.*

For a set of pairs of actions and states M we use M_a to denote the set $\{s \mid (a,s) \in M\}$. Let (S, S^0, Tran) be a finite NAA and let $s \in S$, we then define

$$\Delta_{\mathrm{Tran}}(s) = \bigvee_{M \in \mathrm{Tran}(s)} \Big(\bigwedge_{(a,t) \in M} \langle a \rangle t \wedge \bigwedge_{a \in \Sigma} [a] \Big(\bigvee_{u \in M_a} u \Big) \Big)$$

We then define the νHML formula $nh(S) = (S, S^0, \Delta_{\mathrm{Tran}})$. Notice that variables in $nh(S)$ are states of S.

Lemma 5. *For all NAA S, $S \equiv_{\mathrm{t}} nh(S)$.*

Our translation from νHML to DMTS is based on the constructions in [10]. First, we need a variant of a disjunctive normal form for HML formulae:

Lemma 6. *For any νHML formula (X_1, X_1^0, Δ_1), there exists another formula (X_2, X_2^0, Δ_2) with $[\![\Delta_1]\!] = [\![\Delta_2]\!]$ and such that any formula $\Delta_2(x)$, for $x \in X_2$, is $\mathbf{tt}$ or of the form $\Delta_2(x) = \bigvee_{i\in I} \left(\bigwedge_{j\in J_i} \langle a_{ij}\rangle x_{ij} \wedge \bigwedge_{a\in\Sigma} [a] y_{i,a} \right)$ for finite (possibly empty) index sets I and J_i, $i \in I$, and all $x_{ij}, y_{i,a} \in X_2$. Additionally we can assume that for all $i \in I$, $j \in J_i$, $a \in \Sigma$, $a_{ij} = a$ implies $[\![x_{ij}]\!] \subseteq [\![y_{i,a}]\!]$.*

Let now (X, X^0, Δ) be a νHML formula in the form introduced above, then we define a DMTS $hd(\Delta) = (S, S^0, \dashrightarrow, \longrightarrow)$ as follows:

- $S = \{(x,k) \mid x \in X, \Delta(x) = \bigvee_{i\in I} \phi_i, k \in I \neq \emptyset\} \cup \{\bot, \top\}$,
- $S^0 = \{(x^0, k) \mid x^0 \in X^0\}$.
- For each $(x,k) \in S$ with $\Delta(x) = \bigvee_{i\in I}(\bigwedge_{j\in J_i} \langle a_{ij}\rangle x_{ij} \wedge \bigwedge_{a\in\Sigma} [a] y_{i,a})$ and $I \neq \emptyset$,
  - for each $j \in J_i$, let $\mathrm{Must}_j(x,k) = \{(a_{ij}, (x_{ij}, i')) \in \Sigma \times S\}$,
  - for each $a \in \Sigma$, let $\mathrm{May}_a(x,k) = \{(x', i') \in S \mid [\![x']\!] \subseteq [\![y_{i,a}]\!]\}$.
- Let $\dashrightarrow = \{(s, a, s') \mid s \in S, a \in \Sigma, s' \in \mathrm{May}_a(s)\} \cup \{(\top, a, \top) \mid a \in \Sigma\}$ and $\longrightarrow = \{(s, \mathrm{Must}_j(s)) \mid s = (x,i) \in S, j \in J_i\} \cup \{(\bot, \emptyset)\}$.

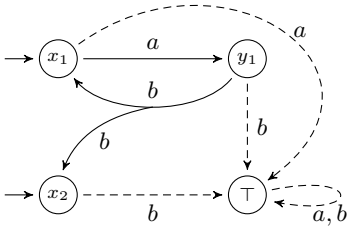
Lemma 7. *For all νHML formulae Δ, $\Delta \equiv_{\mathrm{t}} hd(\Delta)$.*

Further, we remark that the overall translation from DMTS to νHML is quadratic and in the other direction inevitably exponential.

Example 8. Consider the νHML formula $X = (\langle a\rangle(\langle b\rangle X \wedge [a]\mathbf{ff}) \wedge [b]\mathbf{ff}) \vee [a]\mathbf{ff}$. Changing the formula into the normal form of Lemma 6 introduces a new variable Y as illustrated below; X remains the sole initial variable. The translation hd then gives a DMTS with two initial states (the inconsistent state $\bot$ and redundant may transitions such as $x_1 \overset{a}{\dashrightarrow} x_2$, $x_2 \overset{b}{\dashrightarrow} x_1$, etc. have been omitted):

$$X = (\overbrace{\langle a\rangle Y \wedge [a]\mathbf{tt} \wedge [b]\mathbf{ff}}^{x_1}) \vee \underbrace{([a]\mathbf{ff} \wedge [b]\mathbf{tt})}_{x_2}$$

$$Y = \underbrace{\langle b\rangle X \wedge [a]\mathbf{ff} \wedge [b]\mathbf{tt}}_{y_1}$$



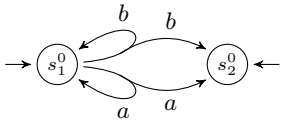
3 Specification Theory

In this section, we introduce operations of conjunction, disjunction, structural composition and quotient for NAA, DMTS and νHML. Together, these operations yield a *complete specification theory* in the sense of [1], which allows for compositional design and verification using both logical and structural operations. We remark that conjunction and disjunction are straightforward for logical formalisms such as νHML, whereas structural composition is more readily defined on behavioural formalisms such as (D)MTS. For the mixed formalism of NAA, disjunction is trivial as we permit multiple initial states, but conjunction requires some work. Note that our construction of conjunction works for nondeterministic systems in contrast to all the work in this area except for [6, 26].

3.1 Disjunction

The disjunction of NAA $S_1 = (S_1, S_1^0, \mathrm{Tran}_1)$ and $S_2 = (S_2, S_2^0, \mathrm{Tran}_2)$ is $S_1 \vee S_2 = (S_1 \cup S_2, S_1^0 \cup S_2^0, \mathrm{Tran}_1 \cup \mathrm{Tran}_2)$. Similarly, the disjunction of two DMTS $S_1 = (S_1, S_1^0, \dashrightarrow_1, \longrightarrow_1)$ and $S_2 = (S_2, S_2^0, \dashrightarrow_2, \longrightarrow_2)$ is $S_1 \vee S_2 = (S_1 \cup S_2, S_1^0 \cup S_2^0, \dashrightarrow_1 \cup \dashrightarrow_2, \longrightarrow_1 \cup \longrightarrow_2)$. It follows that disjunction respects the translation mappings dn and nd from the previous section.

Theorem 9. *Let S_1, S_2, S_3 be NAA or DMTS. Then $[\![S_1 \vee S_2]\!] = [\![S_1]\!] \cup [\![S_2]\!]$. Further, $S_1 \vee S_2 \leq_{\mathrm{m}} S_3$ iff $S_1 \leq_{\mathrm{m}} S_3$ and $S_2 \leq_{\mathrm{m}} S_3$.*



We point out one important distinction between NAA and DMTS: NAA with a *single* initial state are equally expressive as general NAA, while for DMTS, this is not the case. The example on the right shows a DMTS $(S, S^0, \dashrightarrow, \longrightarrow)$, with $S = S^0 = \{s_1^0, s_2^0\}$, $s_1^0 \longrightarrow \{(a, s_1^0), (a, s_2^0)\}$ and $s_1^0 \longrightarrow \{(b, s_1^0), (b, s_2^0)\}$ (and the corresponding may-transitions). Two initial states are necessary for capturing $[\![S]\!]$.

Lemma 10. *For any NAA S there is a NAA $T = (T, T^0, \Psi)$ with $T^0 = \{t^0\}$ a singleton and $S \equiv_{\mathrm{m}} T$.*

3.2 Conjunction

Conjunction for DMTS is an extension of the construction from [6] for multiple initial states. Given two DMTS $(S_1, S_1^0, \dashrightarrow_1, \longrightarrow_1)$, $(S_2, S_2^0, \dashrightarrow_2, \longrightarrow_2)$, we define $S_1 \wedge S_2 = (S, S^0, \dashrightarrow, \longrightarrow)$ with $S = S_1 \times S_2$, $S^0 = S_1^0 \times S_2^0$, and

- $(s_1, s_2) \overset{a}{\dashrightarrow} (t_1, t_2)$ iff $s_1 \overset{a}{\dashrightarrow}_1 t_1$ and $s_2 \overset{a}{\dashrightarrow}_2 t_2$,
- for all $s_1 \longrightarrow N_1$, $(s_1, s_2) \longrightarrow \{(a, (t_1, t_2)) \mid (a, t_1) \in N_1, (s_1, s_2) \overset{a}{\dashrightarrow} (t_1, t_2)\}$,
- for all $s_2 \longrightarrow N_2$, $(s_1, s_2) \longrightarrow \{(a, (t_1, t_2)) \mid (a, t_2) \in N_2, (s_1, s_2) \overset{a}{\dashrightarrow} (t_1, t_2)\}$.

To define conjunction for NAA, we need auxiliary projection functions $\pi_i : \Sigma \times S_1 \times S_2 \to \Sigma \times S_i$. These are defined by

$$\pi_1(M) = \{(a, s_1) \mid \exists s_2 \in S_2 : (a, s_1, s_2) \in M\}$$
$$\pi_2(M) = \{(a, s_2) \mid \exists s_1 \in S_1 : (a, s_1, s_2) \in M\}$$

Given NAA $(S_1, S_1^0, \mathrm{Tran}_1)$, $(S_2, S_2^0, \mathrm{Tran}_2)$, define $S_1 \wedge S_2 = (S, S^0, \mathrm{Tran})$, with $S = S_1 \times S_2$, $S^0 = S_1^0 \times S_2^0$ and $\mathrm{Tran}((s_1, s_2)) = \{M \subseteq \Sigma \times S_1 \times S_2 \mid \pi_1(M) \in \mathrm{Tran}_1(s_1), \pi_2(M) \in \mathrm{Tran}_2(s_2)\}$.

Lemma 11. *For DMTS S_1, S_2, $dn(S_1 \wedge S_2) = dn(S_1) \wedge dn(S_2)$.*

For the translation from NAA to DMTS, $nd(S_1 \wedge S_2) = nd(S_1) \wedge nd(S_2)$ does not necessarily hold, as the translation changes the state space. However, Theorem 12 below will ensure that $nd(S_1 \wedge S_2) \equiv_{\mathrm{t}} nd(S_1) \wedge nd(S_2)$.

Theorem 12. *Let S_1, S_2, S_3 be NAA or DMTS. Then $[\![S_1 \wedge S_2]\!] = [\![S_1]\!] \cap [\![S_2]\!]$. Further, $S_1 \leq_{\mathrm{m}} S_2 \wedge S_3$ iff $S_1 \leq_{\mathrm{m}} S_2$ and $S_1 \leq_{\mathrm{m}} S_3$.*

Theorem 13. *With operations $\wedge$ and $\vee$, the sets of DMTS and NAA form bounded distributive lattices up to $\equiv_{\mathrm{m}}$.*

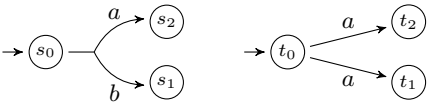
3.3 Structural Composition

We define structural composition for NAA. For NAA $S_1 = (S_1, S_1^0, \mathrm{Tran}_1)$, $S_2 = (S_2, S_2^0, \mathrm{Tran}_2)$, we define $S_1 \| S_2 = (S, S^0, \mathrm{Tran})$ with $S = S_1 \times S_2$, $S^0 = S_1^0 \times S_2^0$, and for all $(s_1, s_2) \in S$, $\mathrm{Tran}((s_1, s_2)) = \{M_1 \| M_2 \mid M_1 \in \mathrm{Tran}_1(s_1), M_2 \in \mathrm{Tran}_2(s_2)\}$, where $M_1 \| M_2 = \{(a, (t_1, t_2)) \mid (a, t_1) \in M_1, (a, t_2) \in M_2\}$.

Lemma 14. *Up to* $\equiv_\mathrm{m}$*, the operator* $\|$ *on NAA is associative and commutative, distributes over* $\vee$*, and has unit* U*, where* U *is the LTS* $(\{s\}, s, \longrightarrow)$ *with* $s \xrightarrow{a} s$ *for all* $a \in \Sigma$*.*

Theorem 15. *For all NAA* S_1*,* S_2*,* S_3*,* S_4*,* $S_1 \leq_\mathrm{m} S_3$ *and* $S_2 \leq_\mathrm{m} S_4$ *imply* $S_1 \| S_2 \leq_\mathrm{m} S_3 \| S_4$*.*

We remark that structural composition on MTS [27] coincides with our NAA composition, so that for MTS S_1, S_2, $dn(S_1) \| dn(S_2) = dn(S_1 \| S_2)$. On the other hand, structural composition for DMTS (with single initial states) as defined in [6] is *weaker* than NAA composition, *i.e.* for DMTS S_1, S_2, and denoting by $\|'$ the composition from [6], only $dn(S_1) \| dn(S_2) \leq_\mathrm{t} dn(S_1 \|' S_2)$ holds. Consider for example the DMTS S and S' in the figure below. When considering their NAA composition, the initial state is the pair (s_0, t_0) with $\mathrm{Tran}((s_0, t_0)) = \{\emptyset, \{(a, (s_2, t_1)), (a, (s_2, t_2))\}$. Since this constraint cannot be represented as a disjunctive must, there is no DMTS with a single initial state which can represent the NAA composition precisely.



Hence the DMTS composition of [6] is a DMTS over-approximation of the NAA composition, and translating from DMTS to NAA before composing (and back again) will generally give a tighter specification. However, as noted already in [24], MTS composition itself is an over-approximation, in the sense that there will generally be implementations $I \in [\![S_1 \| S_2]\!]$ which cannot be written $I = I_1 \| I_2$ for $I_1 \in [\![S_1]\!]$ and $I_2 \in [\![S_2]\!]$; the same is the case for NAA and DMTS.

3.4 Quotient

We now present one of the central contributions of this paper, the construction of quotient. The quotient S/T is to be the most general specification that, when composed with T, refines S. In other words, it must satisfy the property that for all specifications X, $X \leq_\mathrm{m} S/T$ iff $X \parallel T \leq_\mathrm{m} S$. Quotient has been defined for deterministic MTS and for deterministic acceptance automata in [35]; here we extend it to the nondeterministic case (*i.e.* NAA). The construction incurs an exponential blow-up, which however is local and depends on the degree of nondeterminism. We also provide a quotient construction for nondeterministic MTS; this is useful because MTS encodings for NAA can be very compact.

Let $(S, S^0, \mathrm{Tran}_S)$, $(T, T^0, \mathrm{Tran}_T)$ be two NAA. We define the quotient $S/T = (Q, \{q^0\}, \mathrm{Tran}_Q)$. Let $Q = 2^{S\times T}_{\mathrm{Fin}}$ and $q^0 = \{(s^0, t^0) \mid s^0 \in S^0, t^0 \in T^0\}$. States in Q will be written $\{s_1/t_1, \ldots, s_n/t_n\}$ instead of $\{(s_1, t_1), \ldots, (s_n, t_n)\}$.

In the following, we use the notation $x \in\in z$ as a shortcut for the fact that there exists y with $x \in y \in z$. We first define $\mathrm{Tran}_Q(\emptyset) = 2^{\Sigma\times\{\emptyset\}}$. This means that the empty set of pairs is the universal state $\top$. Now let $q = \{s_1/t_1, \ldots, s_n/t_n\} \in Q$. We first define the auxiliary set of possible transitions $pt(q)$ as follows. For $x \in S \cup T$, let $\alpha(x) = \{a \in \Sigma \mid \exists y : (a, y) \in\in \mathrm{Tran}(x)\}$ and $\gamma(q) = \bigcap_i \big(\alpha(s_i) \cup (\Sigma \setminus \alpha(t_i))\big)$. Let further $\pi_a(X) = \{x \mid (a, x) \in X\}$.

Let now $a \in \gamma(q)$. For all $i \in \{1, \ldots, n\}$, let $\{t_{i,1}, \ldots, t_{i,m_i}\} = \pi_a(\bigcup \mathrm{Tran}_T(t_i))$ be the possible next states from t_i after an a-transition, and define

$$pt_a(q) = \big\{\{s_{i,j}/t_{i,j} \mid i \in \{1, \ldots, n\}, j \in \{1, \ldots, m_i\}\} \mid \\ \forall i \in \{1, \ldots, n\} : \forall j \in \{1, \ldots, m_i\} : (a, s_{i,j}) \in\in \mathrm{Tran}_S(s_i)\big\}$$

and $pt(q) = \bigcup_{a\in\Sigma}(\{a\} \times pt_a(q))$. Hence $pt_a(q)$ contains sets of possible next quotient states after an a-transition, each obtained by combining the $t_{i,j}$ with some permutation of possible next a-states in S. We then define

$$\mathrm{Tran}_Q(q) = \{X \subseteq pt(q) \mid \forall i : \forall Y \in \mathrm{Tran}_T(t_i) : X \triangleright Y \in \mathrm{Tran}_S(s_i)\},$$

where the operator $\triangleright$ is defined by $\{s_1/t_1, \ldots, s_k/t_k\} \triangleright t_\ell = s_\ell$ and $X \triangleright Y = \{(a, x \triangleright y) \mid (a, x) \in X, (a, y) \in Y\}$. Hence $\mathrm{Tran}_Q(q)$ contains all sets of (possible) transitions which are compatible with all t_i in the sense that (the projection of) their parallel composition with any set $Y \in \mathrm{Tran}_T(t_i)$ is in $\mathrm{Tran}_S(s_i)$.

Theorem 16. *For all NAA S, T and X, $X \| T \leq_{\mathrm{m}} S$ iff $X \leq_{\mathrm{m}} S/T$.*

Theorem 17. *With operations $\wedge$, $\vee$, $\|$ and $/$, the set of NAA forms a commutative residuated lattice up to $\equiv_{\mathrm{m}}$.*

This theorem makes clear the relation of NAA to linear logic [20]: except for completeness of the lattice induced by $\wedge$ and $\vee$ (*cf.* Theorem 13), NAA form a *commutative unital Girard quantale* [40], the standard algebraic setting for linear logic. Completeness of the lattice can be obtained by allowing infinite conjunctions and disjunctions (and infinite NAA).

3.5 Quotient for MTS

We now give a quotient algorithm for the important special case of MTS, which results in a much more compact quotient than the NAA construction in the previous section. However, MTS are not closed under quotient; *cf.* [28, Thm. 5.5]. We show that the quotient of two MTS will generally be a DMTS.

Let $(S, s^0, \dashrightarrow_S, \longrightarrow_S)$ and $(T, t^0, \dashrightarrow_T, \longrightarrow_T)$ be nondeterministic MTS. We define the quotient $S/T = (Q, \{q^0\}, \dashrightarrow_Q, \longrightarrow_Q)$. We let $Q = 2^{S\times T}_{\mathrm{Fin}}$ as before, and $q^0 = \{(s^0, t^0)\}$. The state $\emptyset \in Q$ is again universal, so we define $\emptyset \overset{a}{\dashrightarrow} \emptyset$ for all $a \in \Sigma$. There are no must transitions from $\emptyset$.

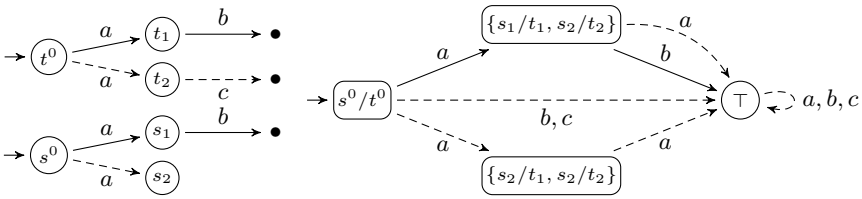


Fig. 4. Two nondeterministic MTS and their quotient

Let $\alpha(s)$, $\gamma(q)$ be as in the previous section. For convenience, we work with sets $\mathrm{May}_a(s)$, for $a \in \Sigma$ and states s, instead of may transitions, *i.e.* we have $\mathrm{May}_a(s) = \{t \mid s \overset{a}{\dashrightarrow} t\}$.

Let $q = \{s_1/t_1, \ldots, s_n/t_n\} \in Q$ and $a \in \Sigma$. First we define the may transitions. If $a \in \gamma(q)$ then for each $i \in \{1, \ldots, n\}$, write $\mathrm{May}_a(t_i) = \{t_{i,1}, \ldots, t_{i,m_i}\}$, and define

$$\mathrm{May}_a(q) = \big\{\{s_{i,j}/t_{i,j} \mid i \in \{1,\ldots,n\}, j \in \{1,\ldots,m_i\}\} \mid \\ \forall i \in \{1,\ldots,n\} : \forall j \in \{1,\ldots,m_i\} : s_{i,j} \in \mathrm{May}_a(s_i)\big\}.$$

For the (disjunctive) must-transitions, we let, for every $s_i \xrightarrow{a} s'$,

$$q \longrightarrow \{(a, M) \in \{a\} \times \mathrm{May}_a(q) \mid \exists t' : s'/t' \in M,\ t_i \xrightarrow{a} t'\}.$$

Example 18. We illustrate the construction on an example. Let S and T be the MTS in the left part of Fig. 4. We construct S/T; the end result is displayed in the right part of the figure.

First we construct the may-successors of s^0/t^0. Under b and c there are no constraints, hence we go to $\top$. For a, we have all permutations of assignments of successors of s to successors of t, namely $\{s_1/t_1, s_1/t_2\}$, $\{s_1/t_1, s_2/t_2\}$, $\{s_2/t_1, s_1/t_2\}$ and $\{s_2/t_1, s_2/t_2\}$. Since there is a must-transition from s (to s_1), we create a disjunctive must-transition to all successors that can be used to yield a must-transition when composed with the must-transition from t to t_1. These are all successors where t_1 is mapped to s_1, hence the first two. However, $\{s_1/t_1, s_1/t_2\}$ will turn out inconsistent, as it requires to refine s_1 by a composition with t_2. As t_2 has no must under b, the composition has none either, hence the must of s_1 can never be matched. As a result, after pruning, the disjunctive must from $\{s^0/t^0\}$ leads only to $\{s_1/t_1, s_2/t_2\}$. Further, $\{s_2/t_1, s_1/t_2\}$ is inconsistent for the same reason, so that we only have one other may-transition under a from $\{s^0/t^0\}$.

Now $\{s_1/t_1, s_2/t_2\}$ is obliged to have a must under b so that it refines s_1 when composed with t_1, but cannot have any c in order to match s_2 when composed with t_2. Similarly, $\{s_2/t_1, s_2/t_2\}$ has neither c nor b. One can easily verify that $T \| (S/T) \equiv_{\mathrm{m}} S$ in this case.

Note that the constructions may create inconsistent states, which have no implementation. In order to get a consistent system, it needs to be pruned. This is standard and the details can be found in [3]. The pruning can be done in polynomial time.

Theorem 19. *For all MTS S, T and X, $X \leq_{\mathrm{m}} S/T$ iff $T \| X \leq_{\mathrm{m}} S$.*

4 Conclusion and Future Work

In this paper we have introduced a general specification framework whose basis consists of three different but equally expressive formalisms: one of a graphical behavioural kind (DMTS), one logic-based (νHML) and one an intermediate language between the former two (NAA). We have shown that the framework possesses a rich algebraic structure that includes logical (conjunction, disjunction) and structural operations (parallel composition and quotient). Moreover, the construction of the quotient solves an open problem in the area of MTS. As for future work, we hope to establish the exact complexity of the quotient constructions. We conjecture that the exponential blow-up of the construction is in general unavoidable.

References

1. Bauer, S.S., David, A., Hennicker, R., Guldstrand Larsen, K., Legay, A., Nyman, U., Wąsowski, A.: Moving from specifications to contracts in component-based design. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 43–58. Springer, Heidelberg (2012)
2. Bauer, S.S., Mayer, P., Legay, A.: MIO workbench: A tool for compositional design with modal input/output interfaces. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 418–421. Springer, Heidelberg (2011)
3. Beneš, N., Delahaye, B., Fahrenberg, U., Křetínský, J., Legay, A.: Hennessy-Milner logic with greatest fixed points as a complete behavioural specification theory. CoRR, abs/1306.0741 (2013)
4. Beneš, N., Křetínský, J.: Process algebra for modal transition systemses. In: MEMICS, pp. 9–18 (2010)
5. Beneš, N., Křetínský, J., Larsen, K.G., Møller, M.H., Srba, J.: Parametric modal transition systems. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 275–289. Springer, Heidelberg (2011)
6. Beneš, N., Černá, I., Křetínský, J.: Modal transition systems: Composition and LTL model checking. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 228–242. Springer, Heidelberg (2011)
7. Blackburn, P.: Representation, reasoning, and relational structures: a hybrid logic manifesto. Logic J. IGPL 8(3), 339–365 (2000)
8. BMoTras, `http://delahaye.benoit.free.fr/BMoTraS.tar`
9. Børjesson, A., Larsen, K.G., Skou, A.: Generality in design and compositional verification using TAV. Formal Meth. Syst. Design 6(3), 239–258 (1995)
10. Boudol, G., Larsen, K.G.: Graphical versus logical specifications. Theor. Comput. Sci. 106(1), 3–20 (1992)
11. Bruns, G.: An industrial application of modal process logic. Sci. Comput. Program. 29(1-2), 3–22 (1997)
12. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 274–287. Springer, Heidelberg (1999)
13. Caires, L., Cardelli, L.: A spatial logic for concurrency (part I). Inf. Comput. 186(2), 194–235 (2003)
14. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logic of Programs. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)

15. Darondeau, P., Dubreil, J., Marchand, H.: Supervisory control for modal specifications of services. In: WODES, pp. 428–435 (2010)
16. D'Ippolito, N., Fischbein, D., Foster, H., Uchitel, S.: MTSA: Eclipse support for modal transition systems construction, analysis and elaboration. In: ETX, pp. 6–10 (2007)
17. Fecher, H., Schmidt, H.: Comparing disjunctive modal transition systems with an one-selecting variant. J. Logic Algebr. Program. 77(1-2), 20–39 (2008)
18. Fecher, H., Steffen, M.: Characteristic mu-calculus formulas for underspecified transition systems. Electr. Notes Theor. Comput. Sci. 128(2), 103–116 (2005)
19. Fokkink, W., van Glabbeek, R.J., de Wind, P.: Compositionality of Hennessy-Milner logic by structural operational semantics. Theor. Comput. Sci. 354(3), 421–440 (2006)
20. Girard, J.-Y.: Linear logic. Theor. Comput. Sci. 50, 1–102 (1987)
21. Hart, J.B., Rafter, L., Tsinakis, C.: The structure of commutative residuated lattices. Internat. J. Algebra Comput. 12(4), 509–524 (2002)
22. Hennessy, M.: Acceptance trees. J. ACM 32(4), 896–928 (1985)
23. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. J. ACM 32(1), 137–161 (1985)
24. Hüttel, H., Larsen, K.G.: The use of static constructs in a modal process logic. In: Logic at Botik, pp. 163–180 (1989)
25. Kozen, D.: Results on the propositional mu-calculus. Theor. Comput. Sci. 27, 333–354 (1983)
26. Larsen, K.G., Liu, X.: Equation solving using modal transition systems. In: LICS, pp. 108–117 (1990)
27. Larsen, K.G.: Modal specifications. In: Automatic Verification Methods for Finite State Systems, pp. 232–246 (1989)
28. Larsen, K.G.: Ideal specification formalism = expressivity + compositionality + decidability + testability + ... In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458, pp. 33–56. Springer, Heidelberg (1990)
29. Larsen, K.G.: Proof systems for satisfiability in Hennessy-Milner logic with recursion. Theor. Comput. Sci. 72, 265–288 (1990)
30. Larsen, K.G., Liu, X.: Compositionality through an operational semantics of contexts. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 526–539. Springer, Heidelberg (1990)
31. Nyman, U.: Modal Transition Systems as the Basis for Interface Theories and Product Lines. PhD thesis, Institut for Datalogi, Aalborg Universitet (2008)
32. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
33. Prior, A.N.: Papers on Time and Tense. Clarendon Press, Oxford (1968)
34. Queille, J.-P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Symp. Program., pp. 337–351 (1982)
35. Raclet, J.-B.: Residual for component specifications. Electr. Notes Theor. Comput. Sci. 215, 93–110 (2008)
36. Raclet, J.-B., Badouel, E., Benveniste, A., Caillaud, B., Passerone, R.: Why are modalities good for interface theories? In: ACSD, pp. 119–127 (2009)
37. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS, pp. 55–74 (2002)
38. Uchitel, S., Chechik, M.: Merging partial behavioural models. In: SIGSOFT FSE, pp. 43–52 (2004)
39. Ward, M., Dilworth, R.P.: Residuated lattices. Trans. AMS 45(3), 335–354 (1939)
40. Yetter, D.N.: Quantales and (noncommutative) linear logic. J. Symb. Log. 55(1), 41–64 (1990)

Merging Partial Behaviour Models with Different Vocabularies

Shoham Ben-David[1], Marsha Chechik[2], and Sebastian Uchitel[3]

[1] Univ. of Waterloo, Canada
[2] Univ. of Toronto, Canada
[3] Univ. of Buenos Aires, Argentina

Abstract. Modal transition systems (MTSs) and their variants such as Disjunctive MTSs (DMTSs) have been extensively studied as a formalism for partial behaviour model specification. Their semantics is in terms of implementations, which are fully specified behaviour models in the form of Labelled Transition Systems. A natural operation for these models is that of *merge*, which should yield a partial model which characterizes all common implementations. Merging has been studied for models with the same vocabularies; however, to enable composition of specifications from different viewpoints, merging of models with *different* vocabularies must be supported as well. In this paper, we first prove that DMTSs are not closed under merge for models with different vocabularies. We then define an extension to DMTS called rDMTS, for which we describe a first exact algorithm for merging partial models, provided they satisfy an easily checkable compatibility condition.

1 Introduction

Behaviour models such as Labelled Transition Systems [10] and Statecharts [8] have been extensively studied as a means to formally describe and analyze behaviour of software systems. These models partition the space of behaviours in two, typically interpreted as *required* behaviour and *prohibited* behaviour. Although notions of refinement for these models have also been studied, limitations in terms of expressiveness have been shown to exist when behavior information is incomplete (e.g., [14]).

Partial behaviour models [3,11,12] allow distinguishing between required, possible and prohibited behaviour, hence supporting partial heterogeneous specifications that include existential (e.g., use-cases) and universal (e.g., safety properties) quantification of behaviour [13]. Refinement involves progressively eliminating possible behaviour until all behaviour is either required or prohibited, as in traditional behaviour models. Indeed, the semantics of partial behaviour models is defined in terms of *implementations*, i.e., two-valued models that provide all the required behaviour of the partial model, and any additional exhibited behaviour is defined as possible.

A key operation on partial models is *composition as conjunction* [16]. That is, given two partial models, it is often desirable to compute a new partial model

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 91–105, 2013.

that captures their common implementations. Such an operation, which we refer to as *model merging*, supports independent development of multiple partial viewpoints that cover different aspects of the intended behavior and subsequent composition into a single model that accurately captures all of these viewpoints.

Partial behaviour model merging has been studied extensively for the case where the models to be merged are defined *on the same vocabulary*. Fischbein and Uchitel [7] showed that Modal Transition Systems (MTSs) [11] are not closed under merge, although the set of common implementations can be represented by a finite set of MTSs. A tool for computing such a merge is described in [4]. Benes et al. [1] showed that a variant of MTSs, known as Disjunctive MTSs (DMTSs) [12], is closed under merge and provided a constructive algorithm for computing such a merge for a set of DMTSs.

Yet, often the partial models to be merged do not completely share their vocabularies. This is especially true in the software engineering context, where different viewpoints are expected to have different scopes and hence different vocabularies. Restricting the merge operation to work only for same-vocabulary models hinders the use of partial models in software engineering contexts.

Attempts at merging partial models defined on different vocabularies have mostly been unsuccessful so far. In [2], Chechik et al. examined the possibility of *embedding* for MTS models. Their idea was to embed each of the models to be merged into a common vocabulary, and then use the same-vocabulary merge algorithm on the results. They show that the embedding idea does not work for MTSs. In [6], Fischbein et al. suggested an *approximation* algorithm to merge MTSs defined on different vocabularies. They present several examples showing that their algorithm is incomplete, but do not try to characterize the subset of models for which the algorithm gives correct results.

In this paper, we approach again the problem of merging partial models defined on different vocabularies. We first prove that DMTSs (and thus MTSs as well) are *not* closed under such merge, which explains why previous attempts were not successful. We then introduce a variant of DMTSs, called *restricted* disjunctive modal transition systems (rDMTSs). Using rDMTSs, we provide the first *exact* algorithm for merging partial models (independently of whether they have the same vocabulary or not). While our algorithm is not complete, we are able to characterize the condition under which the algorithm produces the exact merge. In addition to the standard requirement that the partial models to be merged must be consistent (have a common implementation), we also require that they are *compatible*, i.e., all loops of length greater than one in one model must share some vocabulary with the other model.

Our approach is based on embedding [2]. We show how to embed a model M into a larger vocabulary $\mathcal{A}$, resulting in an rDMTS $M^{\mathcal{A}}$ that preserves the set of implementations of M. Thus, when merging models M and N defined on different vocabularies, we first embed each of them in the union of the vocabularies, and then apply an existing DMTS merge algorithm [1] adapted for rDMTSs. We prove that under the consistency and compatibility conditions, our algorithm produces the exact merge.

The rest of the paper is organized as follows. Sec. 2 gives the background on partial behaviour models. Sec. 3 presents a simple example of two consistent MTSs with a single-letter difference in their vocabularies, and proves that no DMTS can represent exactly the set of their common implementations. Sec. 4 and 5 present the main results of the paper – the rDMTS extension and the new merging algorithm, respectively. We summarize our results and discuss future research directions in Sec. 6. The proofs of most theorems are omitted due to space limitations.

2 Preliminaries

Transition Systems. We start with the concept of *Labelled Transition Systems* (LTSs) [10] which are commonly used for modeling concurrent systems.

Definition 1 (LTS [10]). *A* Labeled Transition System *(LTS) is a structure* (S, L, δ, s_0)*, where* S *is a set of states,* L *is a set of labels,* $\delta \subseteq (S \times L \times S)$ *is the transition relation, and* $s_0 \in S$ *is the initial state.*

Disjunctive Modal Transition Systems (DMTS) [12] are used to specify sets of LTSs. A DMTS distinguishes between two types of transitions – the *possible* and the *disjunctive must*. Transitions that do not appear at all are considered as prohibited. Using a DMTS, one can explicitly model behaviors that are possible in the system and those that the system must exhibit.

Definition 2 (DMTS [12]). *A* Disjunctive Modal Transition System *(DMTS)* M *is a structure* $(S_M, L, \delta^p, \Delta^r, m_0)$*, where* S *is a set of states,* L *is a set of labels,* $\delta^p \subseteq (S_M \times L \times S_M)$ *is the* possible *(or* maybe*) transition relation,* $\Delta^r \subseteq (S_M \times 2^{L \times S_M})$ *is the* disjunctive must *transition relation, and* $m_0 \in S_M$ *is the initial state.*

Modal Transition Systems (MTSs) [11] are a special case of DMTSs where every disjunctive must has exactly one transition.

We use the notation $m \xrightarrow{\ell}_p m'$ to denote a possible transition $(m, \ell, m') \in \delta^p$ ($s \xrightarrow{\ell} s'$ if the model is an LTS). We use $\langle m, V \rangle$ to denote a disjunctive must transition in Δ^r, where V is a set of pairs $V = \{(l_1, m_1), ..., (l_n, m_n)\}$ with $l_1, ..., l_n \in L$ and $m_1, ..., m_n \in S_M$. A disjunct $(l_i, m_i) \in V$ is sometimes called a *leg*, and the entire disjunctive transition – a *DT*. Legs in a DT can also be self-loops. That is, for a DT $\langle s, V \rangle$, there can be legs $(\ell, m') \in V$ s.t. $m' = m$.

We follow [1] to require also that (1) if $\langle m, V \rangle \in \Delta^r$ then V is not empty, and (2) for all $\langle m, V \rangle \in \Delta^r$ and $(\ell, m') \in V$, we have that $(m, \ell, m') \in \delta^p$. That is, there exists a possible transition for every leg in a DT. Graphically, possible transitions are depicted by a question mark: $m \xrightarrow{\ell?} m'$.

A DMTS specifies a set of LTSs – its *implementations*. An LTS I is considered to be a *strong implementation* of a DMTS M if every transition in I is possible in M, and for every DT in M, at least one leg exists in I.

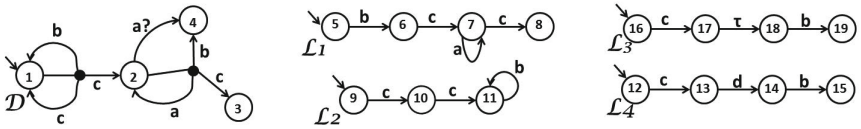


Fig. 1. A DMTS $\mathcal{D}$ and some of its possible implementations. $\mathcal{L}_1$ and $\mathcal{L}_2$ are strong implementations, $\mathcal{L}_3$ is an observational implementation, and $\mathcal{L}_4$ is an alphabet implementation.

Definition 3 (Strong Implementation of DMTSs [12]). *Let* $M = (S_M, L, \delta_M^p, \Delta_M^r, m_0)$ *be a DMTS and* $I = (S_I, L, \delta_I, i_0)$ *be an LTS. We say that* I strongly implements M *if* (m_0, i_0) *is contained in some strong implementation relation* $\mathcal{R} \subseteq S_M \times S_I$*, s.t. if* $(m, i) \in \mathcal{R}$ *then (1)* $\forall (i \xrightarrow{\ell} i')$, $\exists (m \xrightarrow{\ell}_p m')$ *s.t.* $(m', i') \in \mathcal{R}$*; and (2)* $\forall \langle m, V \rangle \in \Delta_M^r$, $\exists (\ell, m') \in V$ *and* $\exists (i \xrightarrow{\ell} i')$ *s.t.* $(m', i') \in \mathcal{R}$.

Example 1. Model $\mathcal{D}$ in Fig. 1 presents a DMTS with one DT on labels b and c and another on a, b and c. Recall that maybe transitions exist for every leg, although they are not explicitly shown. In addition, there is a maybe transition on label a from state 2 to state 4. The LTS $\mathcal{L}_1$ is a strong implementation of $\mathcal{D}$ through the implementation relation $\mathcal{R}_1 = \{(1,5),(1,6),(2,7),(3,8)\}$, and $\mathcal{L}_2$ is also an implementation through $\mathcal{R}_2 = \{(1,9),(1,10),(1,11)\}$.

The set of strong implementations of a DMTS M is denoted by $[[M]]$. Two DMTSs M and N are *consistent* if they have a common implementation, that is, if $[[M]] \cap [[N]] \neq \emptyset$. The *merge* (or "conjunction") of consistent models M and N is a model P s.t. $[[P]] = [[M]] \cap [[N]]$.

Observational and Alphabet Implementations of DMTSs. Hüttel and Larsen [9] were the first to examine MTSs in the presence of *unobservable* labels, denoted by τ, and introduced the notion of *observational* implementations of MTSs. That is, they defined conditions under which an LTS is an implementation of an MTS with unobservable (τ) transitions. Fischbein et al. [5] introduced a more restrictive definition for implementations in the presence of τ's, inspired by branching refinement. This definition was given for MTSs, and we adapt it here to apply to DMTSs. Informally, instead of requiring that a transition from a DT in a DMTS is immediately present in the implementation, as in Def. 3, the observational definition requires that such a transition exists in the implementation, but possibly after a finite (although unbounded) number of τ transitions.

Definition 4 (Observational Implementation of DMTSs). *Let* $M = (S_M, L, \delta_M^p, \Delta_M^r, m_0)$ *be a DMTS and* $I = (S_I, L, \delta_I, i_0)$ *be an LTS. We say that* I *is an* observational implementation *of* M *if* (m_0, i_0) *is contained in some* observational implementation relation $\mathcal{R} \subseteq S_M \times S_I$ *for which the following holds for all* $(m, i) \in \mathcal{R}$*:*

1. $\forall i \xrightarrow{\ell} i'$, *there exists a sequence of possible transitions* $m \xrightarrow{\tau}_p m_1 \xrightarrow{\tau}_p \ldots \xrightarrow{\tau}_p m_j \xrightarrow{\ell}_p m'$, $(m_k, i') \in \mathcal{R}$ *for* $1 \leq k \leq j$, *and* $(m', i') \in \mathcal{R}$.
2. $\forall (m, V) \in \Delta^r_m$, *there exists a sequence of must transitions* $i \xrightarrow{\tau} i_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} i_j \xrightarrow{\ell} i'$ *s.t.* $(m, i_k) \in \mathcal{R}$ *for* $1 \leq k \leq j$, *and* $\exists (\ell, m') \in V$ *s.t.* $(m', i') \in \mathcal{R}$.

In an observational implementation, DMTS and LTS are both defined over the same alphabet, with the addition of the label τ that gets a special treatment. To compare models defined over different alphabets, i.e., to define *observational alphabet implementations*, we follow [15,6], and *hide* labels that are not in the intersection of these alphabets. Hiding is done by replacing such labels by τ's, thus making them unobservable. The resulting models can then be compared using the observational implementation relation (Def. 4).

The definition of hiding in [15] was given for MTSs, and we adapt it to apply to DMTSs, considering every leg of a DT separately.

Definition 5 (Hiding). *Let* $M = (S_M, \alpha M, \delta^p, \Delta^r, m_0)$ *be a DMTS and* X *be a set of labels.* M *with the labels of* X *hidden, denoted* $M \backslash X$, *is a DMTS* $(S_M, \alpha M \backslash X, \delta^{p'}, \Delta^{r'}, m_0)$, *where* $\Delta^{r'}$ *is derived from* Δ^r *by replacing every leg* $(\ell, m') \in V$ *in a DT* $\langle m, V \rangle \in \Delta^r$, *with a leg* (τ, m') *if* $\ell \in X$. *The set* $\delta^{p'}$ *is derived from* δ^p *in the same way, replacing possible transitions* $m \xrightarrow{\ell}_p m'$ *by* $m \xrightarrow{\tau}_p m'$ *if* $\ell \in X$. *For a set of labels* Y, *we use* $M@Y$ *to denote* $M \backslash (\alpha M \backslash Y)$.

Definition 6 (Alphabet Implementation of DMTSs). *An LTS* $I = (S_I, \alpha I, \delta_I, i_0)$ *is an* alphabet implementation *of a DMTS* $M = (S_M, \alpha M, \delta^p_M, \Delta^r_M, m_0)$ *if* $\alpha M \subseteq \alpha I$ *and* $I@\alpha M$ *is an observational implementation of* M.

Example 2. Consider again DMTS $\mathcal{D}$ in Fig. 1. LTS $\mathcal{L}_3$ is an observational implementation of $\mathcal{D}$, via the relation $\{(1, 16), (2, 17), (2, 18), (4, 19)\}$. LTS $\mathcal{L}_4$ is defined on the alphabet $\{a, b, c, d\}$; hiding d results in the model $\mathcal{L}_3$. Thus, $\mathcal{L}_4$ is an alphabet implementation of $\mathcal{D}$.

For a model M with an alphabet αM, let $\mathcal{A}$ be an alphabet such that $\alpha M \subseteq \mathcal{A}$. We denote by $[[M]]_{\mathcal{A}}$ the set of implementations over $\mathcal{A}$ that are alphabet implementations of M.

3 DMTSs Are Not Closed under Alphabet Merge

Our first result explains why the previous attempts to find an alphabet merge algorithm for MTSs have failed. We prove that DMTSs (and thus MTSs as well) are *not closed* under alphabet merge by analyzing the following simple example.

Consider the models in Fig. 2. Model $\mathcal{I}$ has a single must transition labelled c, and we assume its vocabulary is $\{c\}$. We want to merge it with model $\mathcal{J}$ defined over the vocabulary $\{b, c\}$. Thus, we seek a DMTS that specifies exactly all the implementations over the vocabulary $\{b, c\}$ that are common to $\mathcal{I}$ and $\mathcal{J}$. These implementations would be considered "strong" for $\mathcal{J}$, since they share

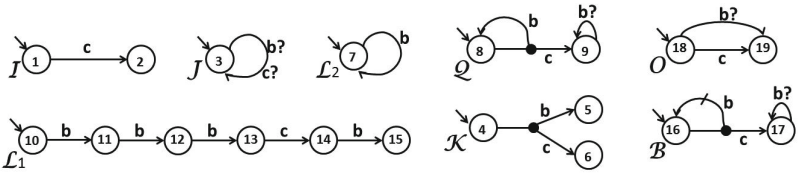


Fig. 2. Models $\mathcal{I}$ and $\mathcal{J}$ do not have a merge in terms of a DMTS. Model $\mathcal{L}_1$ is an example of a common implementation of $\mathcal{I}$ and $\mathcal{J}$. Model $\mathcal{Q}$ is *almost* the merge of $\mathcal{I}$ and $\mathcal{J}$, but the LTS $\mathcal{L}_2$ in an implementation of $\mathcal{Q}$ while not of $\mathcal{I}$. The rDMTS $\mathcal{B}$ is the merge of $\mathcal{I}$ and $\mathcal{J}$. State 4 of model $\mathcal{K}$ is an example of a single-b-allowed state, while state 18 of model $\mathcal{O}$ is not.

the same vocabulary, and "alphabet" for $\mathcal{I}$. The LTS $\mathcal{L}_1$ in Fig. 2 is an example of a possible common implementation: it has a sequence of b transitions followed by one c transition, and then another b transition. It is a strong implementation of $\mathcal{J}$ since $\mathcal{J}$ allows any combination of b's and c's. When all of the b transitions are hidden, it is an observational implementation of $\mathcal{I}$.

Note that the set of common implementations includes all of the implementations that have a finite sequence of b's followed by a c (and possibly more b's after that). Since the length of the b sequence is unbounded (though finite), the number of such implementations is *infinite*. We show that a finite-state DMTS cannot represent such a set.

For our proof, we need the notion of a state from which a single b transition is allowed in an implementation: Let $N = (S_N, \mathcal{A}, \delta_N^p, \Delta_N^r, n_0)$ be a DMTS. A state $s \in S_N$ is a *single-b-allowed* state if there exists an LTS $I = (S_I, \mathcal{A}, \delta_I, i_0)$ strongly implementing N with an implementation relation $\mathcal{R}$, and there exists a state $i \in S_I$, s.t. (1) $(s, i) \in \mathcal{R}$; (2) there exists $i \xrightarrow{b} i' \in \delta_I$; and (3) no other transitions from i exist.

That is, for a state s in N to be a single-b-allowed state, we examine the possible implementations of N. If a legal implementation exists, with a state i corresponding to s (according to the implementation relation), from which a single b transition is departing, then we say that s is a single-b-allowed state. State 4 of model $\mathcal{K}$ in Fig. 2 is a single-b-allowed state, while state 18 of model $\mathcal{O}$ is not, since in every implementation the corresponding state must have a transition on c departing from it.

We now return to prove that no DMTS merge exists for $\mathcal{I}$ and $\mathcal{J}$. Assume by way of contradiction that there exists a DMTS M such that its implementations are exactly all of the implementations common to $\mathcal{I}$ and $\mathcal{J}$. Consider the initial state m_0 of M. The transitions from m_0 must allow an implementation with a single b transition leaving m_0 (such as $\mathcal{L}_1$). This can be achieved, e.g., by a DT as in model $\mathcal{K}$, also allowing an implementation with a single c transition leaving m_0 (such as $\mathcal{I}$ itself, which is a common implementation of $\mathcal{I}$ and $\mathcal{J}$). Note that m_0 *is* a single-b-allowed state, as defined above.

Let us now examine paths in M that contain possible b transitions (we ignore must b transitions, if exist). Let π be the longest path in M, starting from m_0, such that (1) π contains only possible transitions on label b; (2) π does not visit

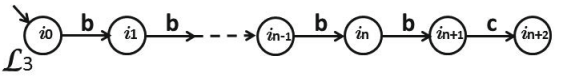


Fig. 3. $\mathcal{L}_3$ is a legal implementation of $\mathcal{I}$ and $\mathcal{J}$ of Fig. 2, showing that a DMTS merge does not exist

a state more than once; and (3) all the states on π are single-b-allowed states. Note that there must be at least one state on π (even if no transition), since the initial state m_0 is a single-b-allowed state. Since M is finite-state, π must be finite. Let n, $n \geq 1$, be the number of states on π and let m_{n-1} denote its final state. Thus, m_{n-1} is a single-b-allowed state, but all the states reachable from m_{n-1} via a transition on b are either not single-b-allowed, or already appear on π. We consider both cases below.

(1) If a b transition from m_{n-1} leads to a state that is already on π, then $[[M]]$ includes an implementation L with a loop on b transitions. But L is not an implementation of $\mathcal{I}$ (since c never appears in L)! Thus, M cannot be the merge of $\mathcal{I}$ and $\mathcal{J}$.

(2) If a b transition from m_{n-1} leads to a state m_n that is not a single-b-allowed state, then either (a) no b transitions can be taken from m_n, or (b) a b transition can be taken from m_n, but only together with another transition (on c or on b or on both). In either case, the implementation $\mathcal{L}_3$ in Fig. 3 does not exist in $[[M]]$, since it includes a path with $n+1$ different single-b-allowed states, while we assumed that the longest such path in M has only n single-b-allowed states. Note though that $\mathcal{L}_3$ is a legal implementation of $\mathcal{I}$ and $\mathcal{J}$. Thus, M cannot be the merge of $\mathcal{I}$ and $\mathcal{J}$.

Based on the above discussion, we conclude the following:

Theorem 1. *DMTSs are not closed under alphabet merge.*

4 Restricted Disjunctive MTS and Embedding

In Sec. 3 we have given a simple example for which an alphabet merge does not exist in the form of a DMTS. This can be fixed by making a small extension to DMTSs.

Consider again models $\mathcal{I}$ and $\mathcal{J}$ of Fig. 2. Model $\mathcal{Q}$ of Fig. 2 is *almost* their merge: it defines all of the legal common implementations of $\mathcal{I}$ and $\mathcal{J}$, but allows one additional implementation, shown as model $\mathcal{L}_2$, with a self-loop on b in the initial state. $\mathcal{L}_2$ is an implementation of $\mathcal{J}$ but not of $\mathcal{I}$ (since no c is ever reached), and thus it is not a common implementation. What if we restrict b in $\mathcal{Q}$ such that the implementation $\mathcal{L}_2$ is ruled out?

In this section, we introduce a new formalism, called *restricted disjunctive MTS (rDMTS)*, that does exactly this. It allows self-looped transition in a DT to be marked as 'restricted'. Model $\mathcal{B}$ in Fig. 2 marks the self-loop on b as restricted. We define a strong implementation relation for rDMTSs that rules out the unwanted implementations, by restricting marked transitions to appear in an implementation only a finite number of times.

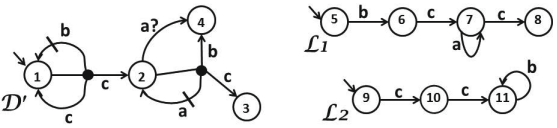


Fig. 4. Model $\mathcal{D}'$ is an rDMTS, $\mathcal{L}_1$ is a strong implementation of it while $\mathcal{L}_2$ is not

We then define the notion of *embedding* that is a key element in our algorithm. Using rDMTS, a model M with an alphabet αM can be embedded (or re-defined) into a larger alphabet $\mathcal{A}$, in a way that preserves alphabet implementations. The embedding procedure is simple: for every DT in M, we add self-looped legs on every letter from $\mathcal{A} \setminus \alpha M$, and mark them as 'restricted'. Model $\mathcal{B}$ in Fig. 2 is the embedding of model $\mathcal{I}$ in the alphabet $\{b, c\}$. We prove that this process preserves implementations, that is, if we let $M^{\mathcal{A}}$ be the re-defined rDMTS, we have that $[[M^{\mathcal{A}}]] = [[M]]_{\mathcal{A}}$. In our example, all of the alphabet implementations of $\mathcal{I}$ over $\{b, c\}$ are also strong implementations of $\mathcal{B}$ and vice versa. Thus, $[[\mathcal{B}]] = [[\mathcal{I}]]_{\{b,c\}}$.

Using rDMTS and the notion of the embedding, we can present our alphabet merge algorithm. Let M and N be models defined over the alphabets αM and αN, respectively, and let $\mathcal{A}$ be the union of the alphabets: $\mathcal{A} = \alpha M \cup \alpha N$. Embedding each model into $\mathcal{A}$ results in rDMTSs $M^{\mathcal{A}}$ and $N^{\mathcal{A}}$ over the *same* alphabet. Two same-alphabet DMTSs can be merged using the algorithm in [1]; we extend it to apply to rDMTSs and prove that if models satisfy a simple *compatibility* condition, our algorithm produces the exact merge.

Our method consists of three main components described below. In Sec. 4.1, we formally define the new formalism, rDMTS, together with a strong implementation relation for it. In Sec. 4.2, we give the *embedding* procedure that preserves alphabet implementations. Finally in Sec. 5, we present an adaptation of the existing strong merge procedure of [1] to work for rDMTSs.

4.1 Restricted Disjunctive MTS

An rDMTS differs from a DMTS in two ways: syntactically – some of the legs of every DT in an rDMTS can be marked as "restricted", and semantically – implementations of a given rDMTS must fulfill additional requirements.

Definition 7 (Restricted DMTS (rDMTS)). *$M = (S, L, \delta^p, \Delta^r, m_0, T)$ is a* restricted DMTS *if $(S, L, \delta^p, \Delta^r, m_0)$ is a DMTS and $T : \Delta^r \longrightarrow 2^{L \times S}$ is a* restricting marking function*, such that for $\langle m, V\rangle \in \Delta^r$, $T(\langle m, V\rangle) \subsetneq V$, and if $(\ell, m') \in T(\langle m, V\rangle)$ then $m' = m$. That is, every restricted leg is a self-loop.*

For an rDMTS M, we denote by $M_\downarrow$ its DMTS part (without the restriction marking). For a DT $\langle m, V\rangle$, we use V_T to denote the set $T(\langle m, V\rangle)$, and call the legs in V_T the *restricted* legs. The non-restricted legs, those in $V \setminus V_T$, are called the *eventual* legs and are denoted by V_E. Note that since $T(\langle m, V\rangle) \subsetneq V$, V_E cannot be empty.

Example 3. Model $\mathcal{D}'$ in Fig. 4 is an example of an rDMTS. Restricted legs are marked with a small line. Note that (a) not all self-loops are necessarily restricted, (b) there can be eventual legs labelled the same way as restricted ones, and (c) different DTs may have differently labelled restricted legs.

We define implementations of rDMTSs by preventing restricted legs from always being picked in a disjunctive transition. That is, we want to ensure that *eventual* legs, that belong to V_E, are eventually picked in every implementation.

Definition 8 (Strong Eventual Implementation). *Let* $M = (S_M, L, \delta^p_M, \delta^r_M, m_0, T)$ *be an rDMTS and* $I = (S_I, L, \delta_I, i_0)$ *be an LTS. I is a* strong eventual implementation *of M if* (m_0, i_0) *is contained in some* strong eventual implementation relation $\mathcal{R} \subseteq S_M \times I_M$*, where* $\mathcal{R}$ *is a strong implementation relation on* $M_\downarrow$ *and I (Def. 3), and for all* $(m, i) \in \mathcal{R}$ *and* $\langle m, V\rangle \in \Delta^r_m$*, there exists a sequence of transitions (called an* eventuality path*)* $i \xrightarrow{\ell_1} i_1 \xrightarrow{\ell_2} ... \xrightarrow{\ell_j} i_j \xrightarrow{\ell} i'$ *in I, s.t. (1)* $\forall 1 \leq k \leq j$*,* $(\ell_k, m) \in V_T$ *and* $(m, i_k) \in \mathcal{R}$*; (2) there exists* m' *with* $(\ell, m') \in V_E$*; and (3)* $(m', i') \in \mathcal{R}$*.*

Example 4. LTS $\mathcal{L}_1$ in Fig. 4 is an implementation of rDMTS $\mathcal{D}'$ via the relation $\mathcal{R}_1 = \{(1,5),(1,6),(2,7),(3,8)\}$. In $\mathcal{L}_1$, a self-loop on state 7 on the restricted a-transition is allowed since an eventuality path from state 7 exists. $\mathcal{L}_2$ is an implementation of $(\mathcal{D}')_\downarrow$ via the relation $\mathcal{R}_2 = \{(1,9),(1,10),(1,11)\}$, but it is not a strong eventual implementation of $\mathcal{D}'$ since there is no eventuality path from state 11.

4.2 rDMTS Embedding

In order to embed a model M into a larger alphabet $\mathcal{A}$, we add self-loop *maybe transitions* on every label from $\mathcal{A} \setminus \alpha M$ to every state of M. In addition, we add self-loop *legs* on every label from $\mathcal{A} \setminus \alpha M$ to every DT in M. We use the restriction function to mark all new legs as restricted. An input to the embedding procedure, formalized in Def. 9, is an rDMTS rather than a DMTS, indicating that an rDMTS can also be embedded into a larger alphabet.

Note that a DMTS can be easily converted into an rDMTS by defining the restriction function T as $T(\langle m, V\rangle) = \emptyset$ for every $\langle m, V\rangle \in \Delta^r$.

Definition 9 (Embedding in a Larger Alphabet). *Let* $M = (S, \alpha M, \delta^p, \Delta^r, m_0, T)$ *be an rDMTS, and* $\mathcal{A}$ *be an alphabet s.t.* $\alpha M \subseteq \mathcal{A}$*. For each state* $m \in S$*, we define a set of "legs" to be added:* $R(m) = \{(\ell, m) \mid \ell \in \mathcal{A} \setminus \alpha M\}$*. An* embedding of M into $\mathcal{A}$ *is an rDMTS* $M^{\mathcal{A}} = (S, \mathcal{A}, \delta^{p'}, \Delta^{r'}, m_0, T')$ *s.t. (1)* $\delta^{p'} = \delta^p \cup \{(m, \ell, m) \mid \ell \in \mathcal{A} \setminus \alpha M\}$*; (2)* $\Delta^{r'} = \{\langle m, V \cup R(m)\rangle \mid \langle m, V\rangle \in \Delta^r\}$*; and (3)* $T'(\langle m, V \cup R(m)\rangle) = T(\langle m, V\rangle) \cup R(m)$*.*

Note that (a) the embedding operation *adds* restricted legs but does not touch existing legs, whether restricted or not, and (b) $(M^{\mathcal{A}})_\downarrow \neq M$ since the embedding procedure adds disjunctive legs as well as maybe transitions, but those are not removed when looking at the DMTS part $(M^{\mathcal{A}})_\downarrow$. The $\downarrow$ operator removes only the restricting markings, leaving the transitions themselves unchanged.

Example 5. Model $\mathcal{I}$ in Fig. 2 is embedded in the alphabet $\{b, c\}$ to get model $\mathcal{B}$ of the same figure. In Fig. 6, $\mathcal{B}$ is further embedded in the alphabet $\{a, b, c\}$ to get model $\mathcal{B}'$.

The above definitions establish that the rDMTS embedding is compositional:

Proposition 1. *Let M be a model and $\mathcal{A}_1, \mathcal{A}_2$ be alphabets s.t. $\alpha M \subseteq \mathcal{A}_1 \subseteq \mathcal{A}_2$. Then, $(M^{\mathcal{A}_1})^{\mathcal{A}_2} = M^{\mathcal{A}_2}$.*

The proof follows directly from Def. 9.

The following theorem guarantees that the embedding procedure constructs an rDMTS such that all alphabet implementations of the original model are strong eventual implementations of the rDMTS. Thus, alphabet implementations of the original DMTS are preserved.

Theorem 2. *Let M be a DMTS and I be an LTS s.t. $\alpha M \subseteq \alpha I$. I is an alphabet implementation of M iff I is a strong eventual implementation of $M^{\alpha I}$.*

Having defined an embedding operation for rDMTSs, models with different vocabularies can be lifted to models with the same vocabularies and merged using a strong merge operator.

5 Merge Using rDMTSs

The merge algorithm for rDMTSs is based on the algorithm of Benes et al. [1] for merging DMTSs defined on the same alphabet. We first review the algorithm given in [1] and then discuss the modifications we need to make so that it applies to rDMTSs.

5.1 Strong Merge of DMTSs

In order for two DMTSs to be merged, the models must be *consistent*, that is, they must have at least one common implementation. The algorithm of [1] is based on a *consistency relation* between the states of the DMTSs to be merged. States m and n are in a consistency relation if for each DT $\langle m, V\rangle$, at least one leg in V has a corresponding possible transition from n and vice versa:

Definition 10 (DMTSs Consistency Relation [1]). *A* strong consistency relation *between DMTSs $M = (S_M, L, \delta^p_M, \Delta^r_M, m_0)$ and $N = (S_N, L, \delta^p_N, \Delta^r_N, n_0)$ is a relation $\mathcal{C} \subseteq S_M \times S_N$ s.t. $(m_0, n_0) \in \mathcal{C}$ and $\forall (m, n) \in \mathcal{C}$, the following holds:*

1. *$\forall \langle m, V\rangle \in \Delta^r_M$, $\exists (l, m') \in V$ and $n \xrightarrow{\ell}_p n'$ in N s.t. $(m', n') \in \mathcal{C}$.*
2. *$\forall \langle n, U\rangle \in \Delta^r_N$, $\exists (q, n') \in U$ and $m \xrightarrow{q}_p m'$ in M s.t. $(m', n') \in \mathcal{C}$.*

Based on a consistency relation $\mathcal{C}$ between M and N, we can now compose them into a single DMTS. The composition is done by constructing, for each DT $\langle m, V\rangle$ in M (or N), a corresponding DT $\langle p, W\rangle$ in the composed model P, where a leg (ℓ, p') exists in W whenever (ℓ, m') exists in V, a leg $n \xrightarrow{\ell}_p n'$ is possible in N, and $(m', n') \in \mathcal{C}$.

Definition 11 (Compose [1]). *Let M and N be DMTSs with the same vocabulary L, and let $\mathcal{C}$ be a consistency relation between them. The $+$ operator between M and N is defined as $[M + N]_{\mathcal{C}} = (\mathcal{C}, L, \delta^p_{M+N}, \Delta^r_{M+N}, (m_0, n_0))$, where δ^p_{M+N} and Δ^r_{M+N} are defined to be the smallest relations that satisfy the following rules:*

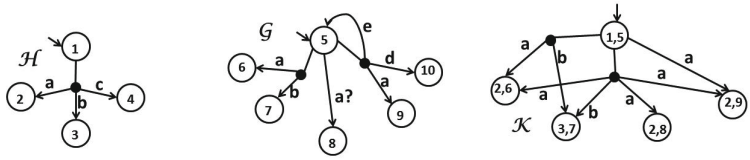


Fig. 5. DMTS $\mathcal{H}$ and $\mathcal{G}$, and their strong merge $\mathcal{K}$

(RM) $\frac{\langle m,V\rangle}{\langle (m,n),W\rangle}$, *where* $W = \{(l,(m',n')) \mid (l,m') \in V \wedge n \xrightarrow{\ell}_p n' \wedge (m',n') \in \mathcal{C}\}$

(MR) $\frac{\langle n,U\rangle}{\langle (m,n),W\rangle}$, *where* $W = \{(l,(m',n')) \mid (l,n') \in U \wedge m \xrightarrow{\ell}_p m' \wedge (m',n') \in \mathcal{C}\}$

(MM) $\frac{m \xrightarrow{\ell}_p m',\ n \xrightarrow{\ell}_p n'}{(m,n) \xrightarrow{\ell}_p (m',n')}$

When $\mathcal{C}$ is the largest consistency relation between M and N, the composition w.r.t. $\mathcal{C}$ becomes the *merge* of M and N.

Theorem 3 (Correctness of Strong DMTS Merge [1]). *Let M and N be DMTSs with the same vocabulary. If $\mathcal{C}$ is the largest consistency relation between the states of M and N then $[M+N]_{\mathcal{C}}$ is the merge of M and N.*

Example 6. Consider the DMTSs $\mathcal{H}$ and $\mathcal{G}$ in Fig. 5, defined over the alphabet $\{a,b,c,d,e\}$. Model $\mathcal{H}$ has one DT with three legs, and $\mathcal{G}$ has two DTs. Rules *MR* and *RM* produce one DT in the merged model $\mathcal{K}$ for each DT in the original models, resulting in three altogether. $\mathcal{H}$'s DT contributes a four-legged DT in $\mathcal{K}$; three of the legs are labelled by a and the fourth – by b. The merged DT is constructed by taking all a-labelled maybe transitions in $\mathcal{G}$ that reach a state consistent with the state 2 of $\mathcal{H}$. In $\mathcal{G}$, there are three such transitions, leading to states 6, 8, and 9 (recall that there is a maybe transition for every leg of a DT), all of which are consistent with state 2 of $\mathcal{H}$. The three-legged DT in $\mathcal{G}$ (on labels a, d and e) results in a DT with a single transition labelled a in $\mathcal{K}$ that reaches the state (2,9) since model $\mathcal{H}$ has no maybe transitions on d or e.

Note that every DT $\langle p,W\rangle$ in the composition P of DMTSs M and N has a *source* DT $\langle m,V\rangle$ in either N or M, and every leg in W has a source leg in V. As discussed above, a DT in P is introduced either by a rule RM or MR, based on a DT that exists in either M or N. The notions of a source DT and source leg are needed in the sequel, and we formalize them below.

Definition 12 (Source DT, Source Leg). *Let M and N be consistent DMTSs, $\mathcal{C}$ be a consistency relation on their states and P be their composition with relation to $\mathcal{C}$. Let $\langle p,W\rangle \in \Delta^r_P$ be a DT in P. We say that $\langle m,V\rangle \in \Delta^r_M$ is the* source DT *of $\langle p,W\rangle$ if (1) $p=(m,n)$ and (2) $(\ell,p') \in W$ iff there exist $(\ell,m') \in V$ and $n \xrightarrow{\ell}_p n'$ in N s.t. $(m',n') \in \mathcal{C}$. We say that a leg $(\ell,m') \in V$ is the* source leg *of $(\ell,p') \in W$ if $p'=(m',n')$.*

5.2 Composition of rDMTSs

We now define the composition of two rDMTSs, M and N. This composition is not necessarily their *merge*, i.e., the set of implementations represented by it

can sometimes include implementations that are not common to M and N. In Sec. 5.3, we characterize the cases for which the composition algorithm yields exactly the merge of M and N.

The composition is obtained by modifying the algorithm given in Def. 11, making it applicable to rDMTSs. We first modify the consistency relation on which the composition is based: we define *eventual* consistency relation, to comply with the definition of strong eventual implementations of rDMTS (Def. 8). We modify the composition itself by adding restriction markings on legs of the composed model.

Definition 13 (Eventual Consistency Relation). *Let* $M = (S_M, L, \delta_M^m, \Delta_M^r, m_0, T_M)$ *and* $N = (S_N, L, \delta_N^m, \Delta_N^r, n_0, T_N)$ *be rDMTSs.* $\mathcal{C}$ *is an* eventual consistency relation *between* M *and* N *if it is a strong consistency relation between* $M_\downarrow$ *and* $N_\downarrow$*, and for all* $(m, n) \in \mathcal{C}$*, the following holds:*

1. $\forall \langle m, V\rangle \in \Delta_M^r$*, there exists a sequence of possible transitions in* N *(called* a possible eventuality path*):* $n \xrightarrow{\ell_1}_p n_1 \xrightarrow{\ell_2}_p \ldots \xrightarrow{\ell_j}_p n_j \xrightarrow{\ell}_p n'$ *s.t. (i)* $\forall 1 \leq i \leq j$, $(\ell_i, m) \in V_T$ *and* $(m, n_i) \in \mathcal{C}$*; (ii) there exists* m' *with* $(\ell, m') \in V_E$*, and (iii)* $(m', n') \in \mathcal{C}$*.*
2. $\forall \langle n, U\rangle \in \Delta_N^r$*, there exists a sequence of possible transitions in* M*:* $m \xrightarrow{q_1}_p m_1 \xrightarrow{q_2}_p \ldots \xrightarrow{q_i}_p m_i \xrightarrow{q}_p m'$ *with the same conditions as above.*

This relation requires the existence of a consistency relation between the DMTS parts $M_\downarrow$ and $N_\downarrow$ (Def. 10). In addition, we need to make sure that for every DT in M (or N), at least one non-restricted leg is eventually allowed on a path in N, and all the transitions in between are restricted. This guarantees the existence of a common implementation. The following theorem states that the opposite is also correct: if a common implementation exists then so does an eventual consistency relation.

Theorem 4. *Let* M *and* N *be rDMSTs. An eventual consistency relation exists between* M *and* N *if and only if there exists an LTS* I *that is a strong eventual implementation of both* M *and* N*.*

The composition of rDMTSs can now be defined. We base it on the largest *eventual* consistency relation $\mathcal{C}$ on the rDMTSs at hand (Def. 13), and use the *source* DT (Def. 12) to mark restricted legs: a self-looped leg in a DT of the composed model is marked as restricted if and only if its source leg is restricted.

Definition 14 (Composition of rDMTSs). *Let* $M = (S_M, L, \delta_M^m, \Delta_M^r, m_0, T_M)$ *and* $N = (S_N, L, \delta_N^m, \Delta_N^r, n_0, T_N)$ *be rDMTSs and let* $\mathcal{C}$ *be the largest eventual consistency relation between them. We define* $P = (S_P, L, \delta_P^m, \Delta_P^r, p_0, T_P)$*, to be the composition of* M *and* N*, where* $P_\downarrow = [M_\downarrow + N_\downarrow]_\mathcal{C}$ *(Def. 11). For each* $\langle p, W\rangle \in \Delta_P^r$*, let* $\langle m, V\rangle \in \Delta_M^r$ *be its* source *DT. We define* $T_P(\langle p, W\rangle) = \{(l, p) \in W \mid \exists (l, m) \in T_M(\langle m, V\rangle)\}$*.*

The rDMTS composition inherits the restriction markings of each DT from its source DT. Note that the rDMTS resulting from a compose operation can be composed again, if desired. We demonstrate this in the example below.

Example 7. Model $\mathcal{B}$ of Fig. 6, defined over the alphabet $\{b, c\}$, is the merge of models $\mathcal{I}$ and $\mathcal{J}$ of Fig 2. We want to compose it further with model $\mathcal{M}$ of Fig. 6, defined

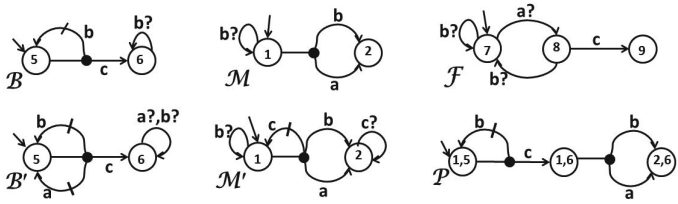


Fig. 6. rDMTSs $\mathcal{M}'$ and $\mathcal{F}$ are consistent with $\mathcal{B}'$ but $\mathcal{F}$ is not compatible with $\mathcal{B}'$. Model $\mathcal{P}$ is the merge of $\mathcal{B}'$ and $\mathcal{M}'$.

over $\{a, b\}$. We thus embed each of the models in the alphabet {a,b,c}: we add a self-loop leg labelled a to the single DT of $\mathcal{B}$, as well as a self-loop possible transition labelled a to state 6. The result is shown as model $\mathcal{B}'$ of Fig. 6. In the same way, we add self-loop transitions labelled c to $\mathcal{M}$, to get model $\mathcal{M}'$. In order to compose them, we find their largest consistency relation $\mathcal{C} = \{(1,5),(1,6),(2,6)\}$. The composition according to Def. 14 is shown in model $\mathcal{P}$.

5.3 Characterizing rDMTS Merge

The composition algorithm given in Def. 14 does not always construct the merge of the input rDMTS. In this section, we define the notion of *compatibility* of two models, and prove that the composition of two compatible rDMTSs is guaranteed to be their merge.

In terms of the original models (before embedding), compatibility means that all loops in one model share some vocabulary with the other model. In the rDMTS terms, it means that there are no loops (of size larger than one) in one model, on labels that are restricted in the other model.

More specifically, let M and N be the rDMTSs to be merged, and let $\mathcal{C}$ be their largest consistency relation. For $(m, n) \in \mathcal{C}$, we require that m does not participate in a loop consisting of labels that are all restricted in some DT from n, and vice versa. Presence of the consistency relation $\mathcal{C}$ allows us to limit this requirement only to pairs of states in $\mathcal{C}$, and thus it is easier to check on rDMTSs rather than on the original models.

We begin by defining a loop on a set of labels.

Definition 15 (A-loops). *Let $M = (S_M, L, \delta_M^m, \Delta_M^r, m_0, T_M)$ be an rDMTS, $m \in S_M$ be a state, and $\mathcal{A}$ be a set of labels. An* $\mathcal{A}$-loop *from m is a sequence of maybe transitions, $m \xrightarrow{\ell_1}_p m_1 \xrightarrow{\ell_2}_p ... \xrightarrow{\ell_j}_p m$, s.t. $\ell_1, ..., \ell_j \in \mathcal{A}$ and $m_1 \neq m$.*

Example 8. Model $\mathcal{F}$ in Fig. 6 has an $\{a, b\}$-loop from state 7.

Using the concept of an $\mathcal{A}$-loop, we now define compatibility between states.

Definition 16 (State Compatibility). *Let $M = (S_M, L, \delta_M^m, \Delta_M^r, m_0, T_M)$ and $N = (S_N, L, \delta_N^m, \Delta_N^r, n_0, T_N)$ be rDMTSs, and $m \in S_M$, $n \in S_N$ be states. Let $\langle m, V\rangle \in \Delta_M^r$, and $\mathcal{A}_T$ be the set of restricted labels in V. If there are no $\mathcal{A}_T$-loops from n then n* is $\langle m, V\rangle$-compatible. *If for all V s.t. $\langle m, V\rangle \in \Delta_M^r$, n is $\langle m, V\rangle$-compatible, then n* is compatible with *m.*

Example 9. Consider models $\mathcal{M}'$ and $\mathcal{B}'$ in Fig. 6. From state 5 of $\mathcal{B}'$, there is only one DT: $(5, \{(a,5),(b,5),(c,6)\})$, with $\mathcal{A}_T = \{a,b\}$. State 1 of $\mathcal{M}'$ has no $\{a,b\}$-loops. Therefore, state 1 of $\mathcal{M}'$ and state 5 of $\mathcal{B}'$ are compatible.

Definition 17 (Model Compatibility). *Let M and N be consistent rDMTSs, with a consistency relation $\mathcal{C}$. M and N are* compatible models with respect to $\mathcal{C}$ *if for all $(m,n) \in \mathcal{C}$, m is compatible with n, and n is compatible with m.*

Example 10. Models $\mathcal{M}'$ and $\mathcal{F}$ in Fig. 6 are consistent with model $\mathcal{B}'$. Yet, $\mathcal{F}$ is not compatible with $\mathcal{B}'$. To see this, note that $(5,7)$, representing the initial states, must exist in every consistency relation between $\mathcal{F}$ and $\mathcal{B}'$. Now consider the loop on labels a and b from state 7 of $\mathcal{F}$. This is an $\{a,b\}$-loop (see Example 8). But $\{a,b\}$ is exactly the set $\mathcal{A}_T$ of the single DT of $\mathcal{B}'$ (see Example 9). Thus, by Def. 17, $\mathcal{F}$ and $\mathcal{B}'$ are not compatible. $\mathcal{M}'$ has no $\{a,b\}$-loops at all and thus is compatible with $\mathcal{B}'$.

The theorem below is one of the main results of our paper, stating correctness of our rDMTS composition operation given in Def. 14.

Theorem 5. *Let M and N be rDMTSs over the same vocabulary and $\mathcal{C}$ be the largest eventual consistency relation between them. Assume that M and N are compatible w.r.t. $\mathcal{C}$ and let P be their composition, as defined by Def. 14. Then P is the strong merge of M and N.*

5.4 Alphabet Merge of Partial Behavioral Models

We now combine the results of Sec. 4 and 5, to form an algorithm for the merge of two models, whether they are LTSs, MTSs, DMTSs or rDMTSs, and whether they are defined on the same alphabet or not.

Algorithm 1 (Alphabet Merge) *Let M and N be models with alphabets αM and αN, respectively, and let $\mathcal{A} = \alpha M \cup \alpha N$. The* alphabet merge *of M and N, denoted by $M +_\alpha N$, is an rDMTS constructed by the following algorithm:*

1. *Construct the embedded models $M^{\mathcal{A}}$ and $N^{\mathcal{A}}$ (Def. 9).*
2. *Compute the largest consistency relation $\mathcal{C}$ on $M^{\mathcal{A}}$ and $N^{\mathcal{A}}$ (Def. 13).*
3. *If $\mathcal{C} = \emptyset$, or if $M^{\mathcal{A}}$ and $N^{\mathcal{A}}$ are not compatible w.r.t. $\mathcal{C}$, return NULL.*
4. *Return the composition of $M^{\mathcal{A}}$ and $N^{\mathcal{A}}$ as defined in Def. 14.*

Theorem 6. *Let P be the result of Algorithm 1 when called on M and N. If P is not NULL, then the set of strong eventual implementations of P is exactly the set of alphabet implementations common to M and N.*

The proof follows immediately from Theorems 2 and 5.

6 Discussion and Future Work

The difficulty in merging models defined over different vocabularies stems from the fact that a common implementation might have to be considered a *strong* implementation of one model and at the same time an *observational* implementation of the other. Restricted DMTSs, introduced in this paper, bridge the

gap between the immediate nature of a strong implementation and the eventual nature of an observational implementation.

A key result of this paper is that rDMTSs are closed under merge for compatible models, which is an important step forward in providing a framework for merging operational yet partial models of system behaviour. We believe the compatibility requirement is sensible from an engineering point of view (models are required to represent viewpoints in which there is a certain degree of overlap). However, experimentation on whether this limitation is inconvenient in practice is necessary. Furthermore, we believe that the compatibility requirement can be relaxed at the cost of making the "restriction marking" function more complex; investigating this further is left for future work.

Acknowledgements. Shoham Ben-David is grateful to the Azrieli Foundation for the award of an Azrieli Fellowship. This work was partially supported by ERC StG PBM-FIMBSE and by the Ontario Ministry of Research and Innovation.

References

1. Beneš, N., Černá, I., Křetínský, J.: Modal Transition Systems: Composition and LTL Model Checking. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 228–242. Springer, Heidelberg (2011)
2. Chechik, M., Brunet, G., Fischbein, D., Uchitel, S.: Partial behavioural models for requirements and early design. In: Proc. of MMOSS 2006 (2006)
3. Dams, D.: Abstract Interpretation and Partition Refinement for Model Checking. PhD thesis, Eindhoven University of Technology, The Netherlands (July 1996)
4. D'Ippolito, N., Fischbein, D., Chechik, M., Uchitel, S.: MTSA: The Modal Transition System Analyser. In: Proc. of ASE 2008, pp. 475–476 (2008)
5. Fischbein, D., Braberman, V.A., Uchitel, S.: A Sound Observational Semantics for Modal Transition Systems. In: Leucker, M., Morgan, C. (eds.) ICTAC 2009. LNCS, vol. 5684, pp. 215–230. Springer, Heidelberg (2009)
6. Fischbein, D., D'Ippolito, N., Brunet, G., Chechik, M., Uchitel, S.: Weak Alphabet Merging of Partial Behavior Models. ACM TOSEM 21(2), 9 (2012)
7. Fischbein, D., Uchitel, S.: On Correct and Complete Strong Merging of Partial Behaviour Models. In: Proc. of SIGSOFT FSE 2008, pp. 297–307 (2008)
8. Harel, D.: StateCharts: A Visual Formalism for Complex Systems. Sc. of Comp. Prog. 8, 231–274 (1987)
9. Hüttel, H., Larsen, K.G.: The Use of Static Constructs in A Modal Process Logic. In: Meyer, A.R., Taitslin, M.A. (eds.) Logic at Botik 1989. LNCS, vol. 363, pp. 163–180. Springer, Heidelberg (1989)
10. Keller, R.M.: Formal Verification of Parallel Programs. CACM 19(7) (1976)
11. Larsen, K.G., Thomsen, B.: A Modal Process Logic. In: Proc. of LICS (1988)
12. Larsen, K.G., Xinxin, L.: Equation Solving Using Modal Transition Systems. In: Proc. of LICS 1990, pp. 108–117 (1990)
13. Sibay, G., Uchitel, S., Braberman, V.A.: Existential Live Sequence Charts Revisited. In: Proc. of ICSE 2008, pp. 41–50 (2008)
14. Uchitel, S., Brunet, G., Chechik, M.: Synthesis of Partial Behavior Models from Properties and Scenarios. IEEE TSE 35(3), 384–406 (2009)
15. Uchitel, S., Chechik, M.: Merging Partial Behavioural Models. In: Proc. of SIGSOFT FSE 2004, pp. 43–52 (2004)
16. Zave, P., Jackson, M.: Conjunction as composition. ACM TOSEM 2, 379–411 (1993)

Solving Parity Games on Integer Vectors

Parosh Aziz Abdulla[1,⋆], Richard Mayr[2,⋆⋆], Arnaud Sangnier[3], and Jeremy Sproston[4,⋆⋆⋆]

[1] Uppsala University, Sweden
[2] University of Edinburgh, UK
[3] LIAFA, Univ Paris Diderot, Sorbonne Paris Cité, CNRS, France
[4] University of Turin, Italy

Abstract. We consider parity games on infinite graphs where configurations are represented by control-states and integer vectors. This framework subsumes two classic game problems: parity games on vector addition systems with states (VASS) and multidimensional energy parity games. We show that the multidimensional energy parity game problem is inter-reducible with a subclass of single-sided parity games on VASS where just one player can modify the integer counters and the opponent can only change control-states. Our main result is that the minimal elements of the upward-closed winning set of these single-sided parity games on VASS are computable. This implies that the Pareto frontier of the minimal initial credit needed to win multidimensional energy parity games is also computable, solving an open question from the literature. Moreover, our main result implies the decidability of weak simulation preorder/equivalence between finite-state systems and VASS, and the decidability of model checking VASS with a large fragment of the modal μ-calculus.

1 Introduction

In this paper, we consider *integer games*: two-player turn-based games where a color (natural number) is associated to each state, and where the transitions allow incrementing and decrementing the values of a finite set of integer-valued counters by constants. We refer to the players as Player 0 and Player 1.

We consider the classical parity condition, together with two different semantics for integer games: the *energy semantics* and the VASS *semantics*. The former corresponds to *multidimensional energy parity games* [7], and the latter to parity games on VASS (a model essentially equivalent to Petri nets [8]). In energy parity games, the winning objective for Player 0 combines a qualitative property, the classical *parity condition*, with a quantitative property, namely the *energy condition*. The latter means that the values of all counters stay above a finite threshold during the entire run of the game. In VASS parity games, the counter values are restricted to *natural numbers*, and in particular any transition that may decrease the value of a counter below zero is disabled (unlike in energy games where such a transition would be immediately winning for Player 1).

⋆ Supported by Uppsala Programming for Multicore Architectures Research Center (UpMarc).
⋆⋆ Supported by Royal Society grant IE110996.
⋆⋆⋆ Supported by the project AMALFI (University of Turin/Compagnia di San Paolo) and the MIUR-PRIN project CINA.

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 106–120, 2013.

So for VASS games, the objective consists only of a parity condition, since the energy condition is trivially satisfied.

We formulate and solve our problems using a generalized notion of game configurations, namely *partial configurations*, in which only a subset C of the counters may be defined. A partial configuration γ denotes a (possibly infinite) set of concrete configurations that are called *instantiations* of γ. A configuration γ' is an instantiation of γ if γ' agrees with γ on the values of the counters in C while the values of counters outside C can be chosen freely in γ'. We declare a partial configuration to be *winning* (for Player 0) if it has an instantiation that is winning. For each decision problem and each set of counters C, we will consider the *C-version* of the problem where we reason about configurations in which the counters in C are defined.

Previous Work. Two special cases of the general C-version are the *abstract* version in which no counters are defined, and the *concrete* version in which all counters are defined. In the energy semantics, the abstract version corresponds to the *unknown initial credit problem* for multidimensional energy parity games, which is coNP-complete [6,7]. The concrete version corresponds to the *fixed initial credit problem*. For energy games without the parity condition, the fixed initial credit problem was solved in [4] (although it does not explicitly mention energy games but instead formulates the problem as a *zero-reachability* objective for Player 1). It follows from [4] that the fixed initial credit problem for d-dimensional energy games can be solved in d-EXPTIME (resp. $(d-1)$-EXPTIME for offsets encoded in unary) and even the upward-closed winning sets can be computed. An EXPSPACE lower bound is derived by a reduction from Petri net coverability. The subcase of one-dimensional energy parity games was considered in [5], where both the unknown and fixed initial credit problems are decidable, and the winning sets (i.e., the minimal required initial energy) can be computed. The assumption of having just one dimension is an important restriction that significantly simplifies the problem. This case is solved using an algorithm which is a generalization of the classical algorithms of McNaughton [13] and Zielonka [16].

However, for general multidimensional energy parity games, computing the winning sets was an open problem, mentioned, e.g., in [6].

In contrast, under the VASS semantics, all these integer game problems are shown to be undecidable for dimensions ≥ 2 in [1], even for simple safety/coverability objectives. (The one-dimensional case is a special case of parity games on one-counter machines, which is PSPACE-complete). A special subcase are *single-sided* VASS games, where just Player 0 can modify counters while Player 1 can only change control-states. This restriction makes the winning set for Player 0 upward-closed, unlike in general VASS games. The paper [14] shows decidability of coverability objectives for single-sided VASS games, using a standard backward fixpoint computation.

Our Contribution. First we show how instances of the single-sided VASS parity game can be reduced to the multidimensional energy parity game, and vice-versa. I.e., energy games correspond to the single-sided subcase of VASS games. Notice that, since parity conditions are closed under complement, it is merely a convention that Player 0 (and not Player 1) is the one that can change the counters.

Our main result is the decidability of single-sided VASS parity games for general partial configurations, and thus in particular for the concrete and abstract versions

described above. The winning set for Player 0 is upward-closed (wrt. the natural multiset ordering on configurations), and it can be computed (i.e., its finitely many minimal elements). Our algorithm uses the Valk-Jantzen construction [15] and a technique similar to Karp-Miller graphs, and finally reduces the problem to instances of the abstract parity problem under the energy semantics, i.e., to the unknown initial credit problem in multidimensional energy parity games, which is decidable by [7].

From the above connection between single-sided VASS parity games and multidimensional energy parity games, it follows that the winning sets of multidimensional energy parity games are also computable. I.e., one can compute the *Pareto frontier* of the minimal initial energy credit vectors required to win the energy parity game. This solves the problem left open in [6,7].

Our results imply further decidability results in the following two areas: semantic equivalence checking and model-checking. Weak simulation preorder between a finite-state system and a general VASS can be reduced to a parity game on a single-sided VASS, and is therefore decidable. Combined with the previously known decidability of the reverse direction [2], this implies decidability of weak simulation equivalence. This contrasts with the undecidability of weak *bisimulation* equivalence between VASS and finite-state systems [11]. The model-checking problem for VASS is decidable for many linear-time temporal logics [10], but undecidable even for very restricted branching-time logics [8]. We show the decidability of model-checking for a restricted class of VASS with a large fragment of the modal μ-calculus. Namely we consider VASS where some states do not perform any updates on the counters, and these states are used to guard the for-all-successors modal operators in this fragment of the μ-calculus, allowing us to reduce the model-checking problem to a parity game on single-sided VASS.

2 Integer Games

Preliminaries. We use $\mathbb{N}$ and $\mathbb{Z}$ to denote the sets of natural numbers (including 0) and integers respectively. For a set A, we define $|A|$ to be the cardinality of A. For a function $f : A \mapsto B$ from a set A to a set B, we use $f[a \leftarrow b]$ to denote the function f' such that $f(a) = b$ and $f'(a') = f(a')$ if $a' \neq a$. If f is partial, then $f(a) = \bot$ means that f is undefined for a. In particular $f[a \leftarrow \bot]$ makes the value of a undefined. We define $dom(f) := \{a |\ f(a) \neq \bot\}$.

Model. We assume a finite set $\mathcal{C}$ of *counters*. An *integer game* is a tuple $\mathcal{G} = \langle Q, T, \kappa \rangle$ where Q is a finite set of *states*, T is a finite set of *transitions*, and $\kappa : Q \mapsto \{0, 1, 2, \ldots, k\}$ is a *coloring* function that assigns to each $q \in Q$ a natural number in the interval $[0..k]$ for some pre-defined k. The set Q is partitioned into two sets Q_0 (states of Player 0) and Q_1 (states of Player 1). A transition $t \in T$ is a triple $\langle q_1, op, q_2 \rangle$ where $q_1, q_2 \in Q$ are states and op is an operation of one of the following three forms (where $c \in \mathcal{C}$ is a counter): (i) c++ increments the value of c by one; (ii) c-- decrements the value of c by one; (iii) *nop* does not change the value of any counter. We define $\texttt{source}(t) = q_1$, $\texttt{target}(t) = q_2$, and $\texttt{op}(t) = op$. We say that $\mathcal{G}$ is *single-sided* in case $op = nop$ for all transitions $t \in T$ with $\texttt{source}(t) \in Q_1$. In other words, in a single-sided game, Player 1 is not allowed to changes the values of the counters, but only the state.

Partial Configurations. A *partial counter valuation* $\vartheta : \mathcal{C} \mapsto \mathbb{Z}$ is a partial function from the set of counters to $\mathbb{Z}$. We also write $\vartheta(c) = \bot$ if $c \notin dom(\vartheta)$. A *partial configuration*

γ is a pair $\langle q, \vartheta \rangle$ where $q \in Q$ is a state and ϑ is a partial counter valuation. We will also consider *nonnegative partial configurations*, where the partial counter valuation takes values in $\mathbb{N}$ instead of $\mathbb{Z}$. We define $\mathtt{state}(\gamma) := q$, $\mathtt{val}(\gamma) := \vartheta$, and $\kappa(\gamma) := \kappa(\mathtt{state}(\gamma))$. We generalize assignments from counter valuations to configurations by defining $\langle q, \vartheta \rangle [c \leftarrow x] = \langle q, \vartheta[c \leftarrow x] \rangle$. Similarly, for a configuration γ and $c \in \mathcal{C}$ we let $\gamma(c) := \mathtt{val}(\gamma)(c)$, $dom(\gamma) := dom(\mathtt{val}(\gamma))$ and $|\gamma| := |dom(\gamma)|$. For a set of counters $C \subseteq \mathcal{C}$, we define $\Theta^C := \{\gamma |\ dom(\gamma) = C\}$, i.e., it is the set of configurations in which the defined counters are exactly those in C. We use Γ^C to denote the restriction of Θ^C to nonnegative partial configurations. We partition Θ^C into two sets Θ_0^C (configurations belonging to Player 0) and Θ_1^C (configurations belonging to Player 1), such that $\gamma \in \Theta_i^C$ iff $dom(\gamma) = C$ and $\mathtt{state}(\gamma) \in Q_i$ for $i \in \{0,1\}$. A configuration is *concrete* if $dom(\gamma) = \mathcal{C}$, i.e., $\gamma \in \Theta^{\mathcal{C}}$ (the counter valuation $\mathtt{val}(\gamma)$ is defined for all counters); and it is *abstract* if $dom(\gamma) = \emptyset$, i.e., $\gamma \in \Theta^{\emptyset}$ (the counter valuation $\mathtt{val}(\gamma)$ is not defined for any counter). In the sequel, we occasionally write Θ instead of $\Theta^{\mathcal{C}}$, and Θ_i instead of $\Theta_i^{\mathcal{C}}$ for $i \in \{0,1\}$. The same notations are defined over nonnegative partial configurations with Γ, and Γ_i^C and Γ_i for $i \in \{0,1\}$. For a nonnegative partial configuration $\gamma = \langle q, \vartheta \rangle \in \Gamma$, and set of counters $C \subseteq \mathcal{C}$ we define the restriction of γ to C by $\gamma' = \gamma | C = \langle q', \vartheta' \rangle$ where $q' = q$ and $\vartheta'(c) = \vartheta(c)$ if $c \in C$ and $\vartheta'(c) = \perp$ otherwise.

Energy Semantics. Under the energy semantics, an integer game induces a transition relation $\longrightarrow_{\mathcal{E}}$ on the set of partial configurations as follows. For partial configurations $\gamma_1 = \langle q_1, \vartheta_1 \rangle$, $\gamma_2 = \langle q_2, \vartheta_2 \rangle$, and a transition $t = \langle q_1, op, q_2 \rangle \in T$, we have $\gamma_1 \xrightarrow{t}_{\mathcal{E}} \gamma_2$ if one of the following three cases is satisfied: (i) $op = c\text{++}$ and either both $\vartheta_1(c) = \perp$ and $\vartheta_2(c) = \perp$ or $\vartheta_1(c) \neq \perp$, $\vartheta_2(c) \neq \perp$ and $\vartheta_2 = \vartheta_1[c \leftarrow \vartheta_1(c) + 1]$; (ii) $op = c\text{--}$, and either both $\vartheta_1(c) = \perp$ and $\vartheta_2(c) = \perp$ or $\vartheta_1(c) \neq \perp$, $\vartheta_2(c) \neq \perp$ and $\vartheta_2 = \vartheta_1[c \leftarrow \vartheta_1(c) - 1]$; (iii) $op = nop$ and $\vartheta_2 = \vartheta_1$. Hence we apply the operation of the transition only if the relevant counter value is defined (otherwise, the counter remains undefined). Notice that, for a partial configuration γ_1 and a transition t, there is at most one γ_2 with $\gamma_1 \xrightarrow{t}_{\mathcal{E}} \gamma_2$. If such a γ_2 exists, we define $t(\gamma_1) := \gamma_2$; otherwise we define $t(\gamma_1) := \perp$. We say that t is *enabled* at γ if $t(\gamma) \neq \perp$. We observe that, in the case of energy semantics, t is not enabled only if $\mathtt{state}(\gamma) \neq \mathtt{source}(t)$.

VASS Semantics. The difference between the energy and VASS semantics is that counters in the case of VASS range over the natural numbers (rather than the integers), i.e. the VASS semantics will be interpreted over nonnegative partial configurations. Thus, the transition relation $\longrightarrow_{\mathcal{V}}$ induced by an integer game $\mathcal{G} = \langle Q, T, \kappa \rangle$ under the VASS semantics differs from the one induced by the energy semantics in the sense that counters are not allowed to assume negative values. Hence $\longrightarrow_{\mathcal{V}}$ is the restriction of $\longrightarrow_{\mathcal{E}}$ to nonnegative partial configurations. Here, a transition $t = \langle q_1, c\text{--}, q_2 \rangle \in T$ is enabled from $\gamma_1 = \langle q_1, \vartheta_1 \rangle$ only if $\vartheta_1(c) > 0$ or $\vartheta_1(c) = \perp$. We assume without restriction that at least one transition is enabled from each partial configuration (i.e., there are no deadlocks) in the VASS semantics (and hence also in the energy semantics). Below, we use $\mathtt{sem} \in \{\mathcal{E}, \mathcal{V}\}$ to distinguish the energy and VASS semantics.

Runs. A *run* ρ in semantics $\mathtt{sem}$ is an infinite sequence $\gamma_0 \xrightarrow{t_1}_{\mathtt{sem}} \gamma_1 \xrightarrow{t_2}_{\mathtt{sem}} \cdots$ of transitions between concrete configurations. A *path* π in $\mathtt{sem}$ is a finite sequence $\gamma_0 \xrightarrow{t_1}_{\mathtt{sem}} \gamma_1 \xrightarrow{t_2}_{\mathtt{sem}} \cdots \gamma_n$ of transitions between concrete configurations. We say that

ρ (resp. π) is a γ-run (resp. γ-path) if $\gamma_0 = \gamma$. We define $\rho(i) := \gamma_i$ and $\pi(i) := \gamma_i$. We assume familiarity with the logic LTL. For an LTL formula ϕ we write $\rho \models_{\mathcal{G}} \phi$ to denote that the run ρ in $\mathcal{G}$ satisfies ϕ. For instance, given a set β of concrete configurations, we write $\rho \models_{\mathcal{G}} \Diamond\beta$ to denote that there is an i with $\gamma_i \in \beta$ (i.e., a member of β eventually occurs along ρ); and write $\rho \models_{\mathcal{G}} \Box\Diamond\beta$ to denote that there are infinitely many i with $\gamma_i \in \beta$ (i.e., members of β occur infinitely often along ρ).

Strategies. A *strategy* of Player $i \in \{0,1\}$ in $\mathtt{sem}$ (or simply an i-strategy in $\mathtt{sem}$) σ_i is a mapping that assigns to each path $\pi = \gamma_0 \xrightarrow{t_1}_{\mathtt{sem}} \gamma_1 \xrightarrow{t_2}_{\mathtt{sem}} \cdots \gamma_n$ with $\mathtt{state}(\gamma_n) \in Q_i$, a transition $t = \sigma_i(\pi)$ with $t(\gamma_n) \neq \bot$ in $\mathtt{sem}$. We use $\Sigma_i^{\mathtt{sem}}$ to denote the sets of i-strategies in $\mathtt{sem}$. Given a concrete configuration γ, $\sigma_0 \in \Sigma_0^{\mathtt{sem}}$, and $\sigma_1 \in \Sigma_1^{\mathtt{sem}}$, we define $\mathtt{run}(\gamma, \sigma_0, \sigma_1)$ to be the unique run $\gamma_0 \xrightarrow{t_1}_{\mathtt{sem}} \gamma_1 \xrightarrow{t_2}_{\mathtt{sem}} \cdots$ such that (i) $\gamma_0 = \gamma$, (ii) $t_{i+1} = \sigma_0(\gamma_0 \xrightarrow{t_1}_{\mathtt{sem}} \gamma_1 \xrightarrow{t_2}_{\mathtt{sem}} \cdots \gamma_i)$ if $\mathtt{state}(\gamma_i) \in Q_0$, and (iii) $t_{i+1} = \sigma_1(\gamma_0 \xrightarrow{t_1}_{\mathtt{sem}} \gamma_1 \xrightarrow{t_2}_{\mathtt{sem}} \cdots \gamma_i)$ if $\mathtt{state}(\gamma_i) \in Q_1$. For $\sigma_i \in \Sigma_i^{\mathtt{sem}}$, we write $[i, \sigma_i, \mathtt{sem}] : \gamma \models_{\mathcal{G}} \phi$ to denote that $\mathtt{run}(\gamma, \sigma_i, \sigma_{1-i}) \models_{\mathcal{G}} \phi$ for all $\sigma_{1-i} \in \Sigma_{1-i}^{\mathtt{sem}}$. In other words, Player i has a *winning strategy*, namely σ_i, which ensures that ϕ will be satisfied regardless of the strategy chosen by Player $1-i$. We write $[i, \mathtt{sem}] : \gamma \models_{\mathcal{G}} \phi$ to denote that $[i, \sigma_i, \mathtt{sem}] : \gamma \models_{\mathcal{G}} \phi$ for some $\sigma_i \in \Sigma_i^{\mathtt{sem}}$.

Instantiations. Two nonnegative partial configurations γ_1, γ_2 are said to be *disjoint* if (i) $\mathtt{state}(\gamma_1) = \mathtt{state}(\gamma_2)$, and (ii) $dom(\gamma_1) \cap dom(\gamma_2) = \emptyset$ (notice that we require the states to be equal). For a set of counters $C \subseteq \mathcal{C}$, and disjoint partial configurations γ_1, γ_2, we say that γ_2 is a *C-complement* of γ_1 if $dom(\gamma_1) \cup dom(\gamma_2) = C$, i.e., $dom(\gamma_1)$ and $dom(\gamma_2)$ form a partitioning of the set C. If γ_1 and γ_2 are disjoint then we define $\gamma_1 \oplus \gamma_2$ to be the nonnegative partial configuration $\gamma := \langle q, \vartheta \rangle$ such that $q := \mathtt{state}(\gamma_1) = \mathtt{state}(\gamma_2)$, $\vartheta(c) := \mathtt{val}(\gamma_1)(c)$ if $\mathtt{val}(\gamma_1)(c) \neq \bot$, $\vartheta(c) := \mathtt{val}(\gamma_2)(c)$ if $\mathtt{val}(\gamma_2)(c) \neq \bot$, and $\vartheta(c) := \bot$ if both $\mathtt{val}(\gamma_1)(c) = \bot$ and $\mathtt{val}(\gamma_2)(c) = \bot$. In such a case, we say that γ is a *C-instantiation* of γ_1. For a nonnegative partial configuration γ we write $[\![\gamma]\!]_C$ to denote the set of C-instantiations of γ. We will consider the special case where $C = \mathcal{C}$. In particular, we say that γ_2 is a *complement* of γ_1 if γ_2 is a $\mathcal{C}$-complement of γ_1, i.e., $\mathtt{state}(\gamma_2) = \mathtt{state}(\gamma_1)$ and $dom(\gamma_1) = \mathcal{C} - dom(\gamma_2)$. We use $\overline{\gamma}$ to denote the set of complements of γ. If $\gamma_2 \in \overline{\gamma_1}$, we say that $\gamma = \gamma_1 \oplus \gamma_2$ is an *instantiation* of γ_1. Notice that γ in such a case is concrete. For a nonnegative partial configuration γ we write $[\![\gamma]\!]$ to denote the set of instantiations of γ. We observe that $[\![\gamma]\!] = [\![\gamma]\!]_C$ and that $[\![\gamma]\!] = \{\gamma\}$ for any concrete nonnegative configuration γ.

Ordering. For nonnegative partial configurations γ_1, γ_2, we write $\gamma_1 \sim \gamma_2$ if $\mathtt{state}(\gamma_1) = \mathtt{state}(\gamma_2)$ and $dom(\gamma_1) = dom(\gamma_2)$. We write $\gamma_1 \sqsubseteq \gamma_2$ if $\mathtt{state}(\gamma_1) = \mathtt{state}(\gamma_2)$ and $dom(\gamma_1) \subseteq dom(\gamma_2)$. For nonnegative partial configurations $\gamma_1 \sim \gamma_2$, we write $\gamma_1 \preceq \gamma_2$ to denote that $\mathtt{state}(\gamma_1) = \mathtt{state}(\gamma_2)$ and $\mathtt{val}(\gamma_1)(c) \leq \mathtt{val}(\gamma_2)(c)$ for all $c \in dom(\gamma_1) = dom(\gamma_2)$. For a nonnegative partial configuration γ, we define $\gamma\uparrow := \{\gamma' \mid \gamma \preceq \gamma'\}$ to be the upward closure of γ, and define $\gamma\downarrow := \{\gamma' \mid \gamma' \preceq \gamma\}$ to be the downward closure of γ. Notice that $\gamma\uparrow = \gamma\downarrow = \{\gamma\}$ for any abstract configuration γ. For a set $\beta \subseteq \Gamma^C$ of nonnegative partial configurations, let $\beta\uparrow := \cup_{\gamma \in \beta} \gamma\uparrow$. We say that β is upward-closed if $\beta\uparrow = \beta$. For an upward-closed set $\beta \subseteq \Gamma^C$, we use $min(\beta)$ to denote the (by Dickson's Lemma unique and finite) set of minimal elements of β.

Winning Sets of Partial Configurations. For a nonnegative partial configuration γ, we write $[i,\mathtt{sem}] : \gamma \models_{\mathcal{G}} \phi$ to denote that $\exists \gamma' \in [\![\gamma]\!].[i,\mathtt{sem}] : \gamma' \models_{\mathcal{G}} \phi$, i.e., Player i is winning from some instantiation γ' of γ. For a set $C \subseteq \mathcal{C}$ of counters, we define $\mathcal{W}[\mathcal{G},\mathtt{sem},i,C](\phi) := \{\gamma \in \Gamma^C \mid [\mathtt{sem},i] : \gamma \models_{\mathcal{G}} \phi\}$. If $\mathcal{W}[\mathcal{G},\mathtt{sem},i,C](\phi)$ is upward-closed, we define the *Pareto frontier* as $\mathtt{Pareto}[\mathcal{G},\mathtt{sem},i,C](\phi) := min(\mathcal{W}[\mathcal{G},\mathtt{sem},i,C](\phi))$.

Properties. We show some useful properties of the ordering on nonnegative partial configurations. Note that for nonnegative partial configurations, we will not make distinctions between the energy semantics and the VASS semantics; this is due to the fact that in nonnegative partial configurations and in their instantiations we only consider positive values for the counters. For the energy semantics, as we shall see, this will not be a problem since we will consider winning runs where the counter never goes below 0. We now show *monotonicity* and (under some conditions) "reverse monotonicity" of the transition relation wrt. $\preceq$. We write $\gamma_1 \longrightarrow_{\mathtt{sem}} \gamma_2$ if there exists t such that $\gamma_1 \xrightarrow{t}_{\mathtt{sem}} \gamma_2$.

Lemma 1. *Let γ_1, γ_2, and γ_3 be nonnegative partial configurations. If (i) $\gamma_1 \longrightarrow_{\mathcal{V}} \gamma_2$, and (ii) $\gamma_1 \preceq \gamma_3$, then there is a γ_4 such that $\gamma_3 \longrightarrow_{\mathcal{V}} \gamma_4$ and $\gamma_2 \preceq \gamma_4$. Furthermore, if (i) $\gamma_1 \longrightarrow_{\mathcal{V}} \gamma_2$, and (ii) $\gamma_3 \preceq \gamma_1$, and (iii) $\mathcal{G}$ is single-sided and (iv) $\gamma_1 \in \Gamma_1$, then there is a γ_4 such that $\gamma_3 \longrightarrow_{\mathcal{V}} \gamma_4$ and $\gamma_4 \preceq \gamma_2$.*

We consider a version of the *Valk-Jantzen* lemma [15], expressed in our terminology.

Lemma 2. *[15] Let $C \subseteq \mathcal{C}$ and let $U \subseteq \Gamma^C$ be upward-closed. Then, $min(U)$ is computable if and only if, for any nonnegative partial configuration γ with $dom(\gamma) \subseteq C$, we can decide whether $[\![\gamma]\!]_C \cap U \neq \emptyset$.*

3 Game Problems

Here we consider the parity winning condition for the integer games defined in the previous section. First we establish a correspondence between the VASS semantics when the underlying integer game is single-sided, and the energy semantics in the general case. We will show how instances of the single-sided VASS parity game can be reduced to the energy parity game, and vice-versa. Figure 1 depicts a summary of our results. For either semantics, an instance of the problem consists of an integer game $\mathcal{G}$ and a partial configuration γ. For a given set of counters $C \subseteq \mathcal{C}$, we will consider the *C-version* of the problem where we assume that $dom(\gamma) = C$. In particular, we will consider two special cases: (i) the *abstract* version in which we assume that γ is abstract (i.e., $dom(\gamma) = \emptyset$), and (ii) the *concrete* version in which we assume that γ is concrete (i.e., $dom(\gamma) = \mathcal{C}$). The abstract version of a problem corresponds to the *unknown initial credit problem* [6,7], while the concrete one corresponds to deciding if a given initial credit is sufficient or, more generally, computing the Pareto frontier (left open in [6,7]).

Winning Conditions. Assume an integer game $\mathcal{G} = \langle Q, T, \kappa \rangle$ where $\kappa : Q \mapsto \{0,1,2,\ldots,k\}$. For a partial configuration γ and $i : 0 \leq i \leq k$, the relation $\gamma \models_{\mathcal{G}} (\mathtt{color} = i)$ holds if $\kappa(\mathtt{state}(\gamma)) = i$. The formula simply checks the color of the state of γ. The formula $\gamma \models_{\mathcal{G}} \overline{\mathtt{neg}}$ holds if $\mathtt{val}(\gamma)(c) \geq 0$ for all $c \in dom(\gamma)$.

The formula states that the values of all counters are nonnegative in γ. For $i : 0 \leq i \leq k$, the predicate $even(i)$ holds if i is even. Define the path formula $\texttt{Parity} := \bigvee_{(0 \leq i \leq k) \wedge even(i)} \left((\Box\Diamond(\texttt{color} = i)) \wedge \left(\bigwedge_{i < j \leq k} \Diamond\Box\neg(\texttt{color} = j) \right) \right)$. The formula states that the highest color that appears infinitely often along the path is even.

Energy Parity. Given an integer game $\mathcal{G}$ and a partial configuration γ, we ask whether $[0, \mathcal{E}] : \gamma \models_{\mathcal{G}} \texttt{Parity} \wedge (\Box\overline{\texttt{neg}})$, i.e., whether Player 0 can force a run in the energy semantics where the parity condition is satisfied and at the same time the counters remain nonnegative. The abstract version of this problem is equivalent to the unknown initial credit problem in classical energy parity games [6,7], since it amounts to asking for the *existence* of a threshold for the initial counter values from which Player 0 can win. The nonnegativity objective $(\Box\overline{\texttt{neg}})$ justifies our restriction to nonnegative partial configurations in our definition of the instantiations and hence of the winning sets.

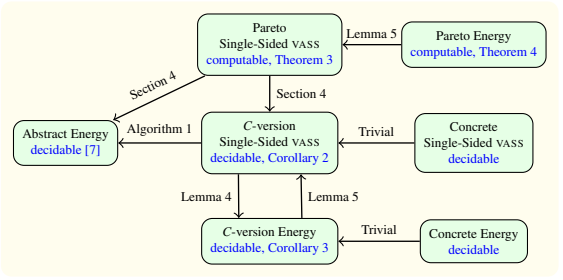


Fig. 1. Problems considered in the paper and their relations. For each property, we state the lemma that show its decidability/computability. The arrows show the reductions of problem instances that we show in the paper.

Theorem 1. *[7] The abstract energy parity problem is decidable.*

The winning set $\mathcal{W}[\mathcal{G}, \mathcal{E}, 0, C](\texttt{Parity} \wedge \Box\overline{\texttt{neg}})$ is upward-closed for $C \subseteq \mathcal{C}$. Intuitively, if Player 0 can win the game with a certain value for the counters, then any higher value for these counters also allows him to win the game with the same strategy. This is because both the possible moves of Player 1 and the colors of configurations depend only on the control-states.

Lemma 3. *For any $C \subseteq \mathcal{C}$, the set $\mathcal{W}[\mathcal{G}, \mathcal{E}, 0, C](\texttt{Parity} \wedge \Box\overline{\texttt{neg}})$ is upward-closed.*

Since this winning set is upward-closed, it follows from Dickson's Lemma that it has finitely many minimal elements. These minimal elements describe the Pareto frontier of the minimal initial credit needed to win the game. In the sequel we will show how to compute this set $\texttt{Pareto}[\mathcal{G}, \mathcal{E}, 0, C](\texttt{Parity} \wedge \Box\overline{\texttt{neg}})) := min(\mathcal{W}[\mathcal{G}, \mathcal{E}, 0, C](\texttt{Parity} \wedge \Box\overline{\texttt{neg}}))$; cf. Theorem 4.

VASS Parity. Given an integer game $\mathcal{G}$ and a nonnegative partial configuration γ, we ask whether $[0, \mathcal{V}] : \gamma \models_{\mathcal{G}} \texttt{Parity}$, i.e., whether Player 0 can force a run in the VASS semantics where the parity condition is satisfied. (The condition $\Box\overline{\texttt{neg}}$ is always trivially satisfied in VASS.) In general, this problem is undecidable as shown in [1], even for simple coverability objectives instead of parity objectives.

Theorem 2. *[1] The VASS Parity Problem is undecidable.*

We will show that decidability of the VASS parity problem is regained under the assumption that $\mathcal{G}$ is single-sided. In [14] it was already shown that, for a single-sided VASS game with reachability objectives, it is possible to compute the set of winning configurations. However, the proof for parity objectives is much more involved.

Correspondence of Single-Sided VASS *Games and Energy Games.* We show that single-sided VASS parity games can be reduced to energy parity games, and vice-versa. The following lemma shows the direction from VASS to energy.

Lemma 4. *Let $\mathcal{G}$ be a single-sided integer game and let γ be a nonnegative partial configuration. Then $[0,\mathcal{V}]:\gamma \models_{\mathcal{G}} \texttt{Parity}$ iff $[0,\mathcal{E}]:\gamma \models_{\mathcal{G}} \texttt{Parity} \wedge \Box\overline{\texttt{neg}}$.*

Hence for a single-sided $\mathcal{G}$ and any set $C \subseteq \mathcal{C}$, we have $\mathcal{W}[\mathcal{G},\mathcal{V},0,C](\texttt{Parity}) = \mathcal{W}[\mathcal{G},\mathcal{E},0,C](\texttt{Parity} \wedge \Box\overline{\texttt{neg}})$. Consequently, using Lemma 3 and Theorem 1, we obtain the following corollary.

Corollary 1. *Let $\mathcal{G}$ be single-sided and $C \subseteq \mathcal{C}$.*

1. *$\mathcal{W}[\mathcal{G},\mathcal{V},0,C](\texttt{Parity})$ is upward-closed.*
2. *The C-version single-sided* VASS *parity problem is reducible to the C-version energy parity problem.*
3. *The* abstract *single-sided* VASS *parity problem (i.e., where $C = \emptyset$) is decidable.*

The following lemma shows the reverse reduction from energy parity games to single-sided VASS parity games.

Lemma 5. *Given an integer game $\mathcal{G} = \langle Q, T, \kappa\rangle$, one can construct a single-sided integer game $\mathcal{G}' = \langle Q', T', \kappa'\rangle$ with $Q \subseteq Q'$ such that $[0,\mathcal{E}]:\gamma \models_{\mathcal{G}} \texttt{Parity} \wedge \Box\overline{\texttt{neg}}$ iff $[0,\mathcal{V}]:\gamma \models_{\mathcal{G}'} \texttt{Parity}$ for every nonnegative partial configuration γ of $\mathcal{G}$.*

Proof sketch. Since $\mathcal{G}'$ needs to be single-sided, Player 1 cannot change the counters. Thus the construction forces Player 0 to simulate the moves of Player 1. Whenever a counter drops below zero in $\mathcal{G}$ (and thus Player 0 loses), Player 0 cannot perform this simulation in $\mathcal{G}'$ and is forced to go to a losing state instead. □

Computability Results. The following theorem (shown in Section 4) states our main computability result. For single-sided VASS parity games, the minimal elements of the winning set $\mathcal{W}[\mathcal{G},\mathcal{V},0,C](\texttt{Parity})$ (i.e., the Pareto frontier) are computable.

Theorem 3. *If $\mathcal{G}$ is single-sided then $\texttt{Pareto}[\mathcal{G},\mathcal{V},0,C](\texttt{Parity})$ is computable.*

In particular, this implies decidability.

Corollary 2. *For any set of counters $C \subseteq \mathcal{C}$, the C-version single-sided* VASS *parity problem is decidable.*

From Theorem 3 and Lemma 5 we obtain the computability of the Pareto frontier of the minimal initial credit needed to win general energy parity games.

Theorem 4. $\texttt{Pareto}[\mathcal{G},\mathcal{E},0,C](\texttt{Parity} \wedge \Box\overline{\texttt{neg}})$ *is computable for any game $\mathcal{G}$.*

Corollary 3. *The C-version energy parity problem is decidable.*

4 Solving Single-Sided VASS Parity Games (Proof of Theorem 3)

Consider a single-sided integer game $\mathcal{G} = \langle Q, T, \kappa \rangle$ and a set $C \subseteq \mathcal{C}$ of counters. We will show how to compute the set $\texttt{Pareto}[\mathcal{G}, \mathcal{V}, 0, C](\texttt{Parity})$. We reduce the problem of computing the Pareto frontier in the single-sided VASS parity game to solving the abstract energy parity game problem, which is decidable by Theorem 1.

We use induction on $k = |C|$. As we shall see, the base case is straightforward. We perform the induction step in two phases. First we show that, under the induction hypothesis, we can reduce the problem of computing the Pareto frontier to the problem of solving the C-version single-sided VASS parity problem (i.e., we need only to consider individual nonnegative partial configurations in Γ^C). In the second phase, we introduce an algorithm that translates the latter problem to the abstract energy parity problem.

Base Case. Assume that $C = \emptyset$. In this case we are considering the abstract single-sided VASS parity problem. Recall that $\gamma \uparrow = \{\gamma\}$ for any γ with $dom(\gamma) = \emptyset$. Since $C = \emptyset$, it follows that $\texttt{Pareto}[\mathcal{G}, \mathcal{V}, 0, C](\texttt{Parity}) = \{\gamma \mid (dom(\gamma) = \emptyset) \wedge ([0, \mathcal{V}] : \gamma \models_{\mathcal{G}} \texttt{Parity})\}$. In other words, computing the Pareto frontier in this case reduces to solving the abstract single-sided VASS parity problem, which is decidable by Corollary 1.

From Pareto Sets to VASS *Parity.* Assuming the induction hypothesis, we reduce the problem of computing the set $\texttt{Pareto}[\mathcal{G}, \mathcal{V}, 0, C](\texttt{Parity})$ to the C-version single-sided VASS parity problem, i.e., the problem of checking whether $[0, \mathcal{V}] : \gamma \models_{\mathcal{G}} \texttt{Parity}$ for some $\gamma \in \Gamma^C$ when the underlying integer game is single-sided. To do that, we will instantiate the Valk-Jantzen lemma as follows. We instantiate $U \subseteq \Gamma^C$ in Lemma 2 to be $\mathcal{W}[\mathcal{G}, \mathcal{V}, 0, C](\texttt{Parity})$ (this set is upward-closed by Corollary 1 since $\mathcal{G}$ is single-sided). Take any nonnegative partial configuration γ with $dom(\gamma) \subseteq C$. We consider two cases. First, if $dom(\gamma) = C$, then we are dealing with the C-version single-sided VASS parity game which will show how to solve in the sequel. Second, consider the case where $dom(\gamma) = C' \subset C$. By the induction hypothesis, we can compute the (finite) set $\texttt{Pareto}[\mathcal{G}, \mathcal{V}, 0, C'](\texttt{Parity}) = min(\mathcal{W}[\mathcal{G}, \mathcal{V}, 0, C'](\texttt{Parity}))$. Then to solve this case, we use the following lemma.

Lemma 6. *For all nonnegative partial configurations γ such that $dom(\gamma) = C' \subset C$, we have $[\![\gamma]\!]_C \cap \mathcal{W}[\mathcal{G}, \mathcal{V}, 0, C](\texttt{Parity}) \neq \emptyset$ iff $\gamma \in \mathcal{W}[\mathcal{G}, \mathcal{V}, 0, C'](\texttt{Parity})$.*

Hence checking $[\![\gamma]\!]_C \cap \mathcal{W}[\mathcal{G}, \mathcal{V}, 0, C](\texttt{Parity}) \neq \emptyset$ amounts to simply comparing γ with the elements of the finite set $\texttt{Pareto}[\mathcal{G}, \mathcal{V}, 0, C'](\texttt{Parity})$, because $\mathcal{W}[\mathcal{G}, \mathcal{V}, 0, C'](\texttt{Parity})$ is upward-closed by Corollary 1.

From VASS *Parity to Abstract Energy Parity.* We introduce an algorithm that uses the induction hypothesis to translate an instance of the C-version single-sided VASS parity problem to an equivalent instance of the abstract energy parity problem.

The following definition and lemma formalize some consequences of the induction hypothesis. First we define a relation that allows us to directly classify some nonnegative partial configurations as winning for Player 1 (resp. Player 0).

Definition 1. *Consider a nonnegative partial configuration γ and a set of nonnegative partial configurations β. We write $\beta \lhd \gamma$ if: (i) for each $\hat{\gamma} \in \beta$, $dom(\hat{\gamma}) \subseteq C$ and $|\gamma| = |\hat{\gamma}| + 1$, and (ii) for each $c \in dom(\gamma)$ there is a $\hat{\gamma} \in \beta$ such that $\hat{\gamma} \preceq \gamma[c \leftarrow \bot]$.*

Lemma 7. *Let* $\beta = \bigcup_{C' \subseteq C, |C'|=|C|-1} \texttt{Pareto}[\mathcal{G}, \mathcal{V}, 0, C'](\texttt{Parity})$ *be the Pareto frontier of minimal Player* 0 *winning nonnegative partial configurations with one counter in C undefined. Let* $\{c_i, \ldots, c_j\} = \mathcal{C} - C$ *be the counters outside C.*

1. *For every* $\hat{\gamma} \in \beta$ *with* $\{c\} = C - dom(\hat{\gamma})$ *there exists a minimal finite number* $v(\hat{\gamma})$ *s.t.* $[\![\hat{\gamma}[c \leftarrow v(\hat{\gamma})]]\!] \cap \mathcal{W}[\mathcal{G}, \mathcal{V}, 0, \mathcal{C}](\texttt{Parity}) \neq \emptyset$.
2. *For every* $\hat{\gamma} \in \beta$ *there is a number* $u(\hat{\gamma})$ *s.t.* $\hat{\gamma}[c \leftarrow v(\hat{\gamma})][c_i \leftarrow u(\hat{\gamma}), \ldots, c_j \leftarrow u(\hat{\gamma})] \in \mathcal{W}[\mathcal{G}, \mathcal{V}, 0, \mathcal{C}](\texttt{Parity})$*, i.e., assigning value* $u(\hat{\gamma})$ *to counters outside C is sufficient to make the nonnegative configuration winning for Player* 0.
3. *If* $\gamma \in \Gamma^C$ *is a Player* 0 *winning nonnegative partial configuration, i.e.,* $[\![\gamma]\!] \cap \mathcal{W}[\mathcal{G}, \mathcal{V}, 0, \mathcal{C}](\texttt{Parity}) \neq \emptyset$*, then* $\beta \lhd \gamma$.

The third part of this lemma implies that if $\neg(\beta \lhd \gamma)$ then we can directly conclude that γ is not winning for Player 0 (and thus winning for Player 1) in the parity game.

Now we are ready to present the algorithm (Algorithm 1).

Input and output of the algorithm. The algorithm inputs a single-sided integer game $\mathcal{G} = \langle Q, T, \kappa \rangle$, and a nonnegative partial configuration γ where $dom(\gamma) = C$. To check whether $[0, \mathcal{V}] : \gamma \models_{\mathcal{G}} \texttt{Parity}$, it constructs an instance of the abstract energy parity problem. This instance is defined by a new integer game $\mathcal{G}^{out} = \langle Q_{out}, T_{out}, \kappa_{out} \rangle$ with counters in $\mathcal{C} - C$, and a nonnegative partial configuration γ^{out}. Since we are considering the abstract version of the problem, the configuration γ^{out} is of the form $\gamma^{out} = \langle q^{out}, \vartheta_{out} \rangle$ where $dom(\vartheta_{out}) = \emptyset$. The latter property means that γ^{out} is uniquely determined by the state q^{out} (all counter values are undefined). Lemma 9 relates $\mathcal{G}$ with the newly constructed $\mathcal{G}^{out}$.

Algorithm 1. Building an instance of the abstract energy parity problem.

Input: $\mathcal{G} = \langle Q, T, \kappa \rangle$: Single-Sided Integer Game; $\gamma \in \Gamma^C$ with $|C| = k > 0$.
Output: $\mathcal{G}^{out} = \langle Q^{out}, T^{out}, \kappa^{out} \rangle$: integer game;
$q^{out} \in Q^{out}$; $\gamma^{out} = \langle q^{out}, \vartheta_{out} \rangle$ where $dom(\vartheta_{out}) = \emptyset$; $\lambda : Q_{out} \cup T_{out} \mapsto \Gamma^C \cup T$

$\beta \leftarrow \bigcup_{(C' \subseteq C) \wedge (|C'| = |C|-1)} \texttt{Pareto}[\mathcal{G}, \mathcal{V}, 0, C'](\texttt{Parity})$;
$T^{out} \leftarrow \emptyset$; $\texttt{new}(q^{out})$; $\kappa(q^{out}) \leftarrow \kappa(\gamma)$; $\lambda(q^{out}) \leftarrow \gamma$; $Q_{out} \leftarrow \{q^{out}\}$;
if $\lambda(q^{out}) \in \Gamma_0$ **then** $Q_0^{out} \leftarrow \{q^{out}\}$; $Q_1^{out} \leftarrow \emptyset$ **else** $Q_1^{out} \leftarrow \{q^{out}\}$; $Q_0^{out} \leftarrow \emptyset$;
$\texttt{ToExplore} \leftarrow \{q^{out}\}$;
while $\texttt{ToExplore} \neq \emptyset$ **do**
  Pick and remove a $q \in \texttt{ToExplore}$;
  if $\neg(\beta \lhd \lambda(q))$ **then**
    $\kappa^{out}(q) \leftarrow 1$; $T^{out} \leftarrow T^{out} \cup \{\langle q, nop, q \rangle\}$
  else if $\exists q'.(q', q) \in (T^{out})^* \wedge (\lambda(q') \prec \lambda(q))$ **then**
    $\kappa^{out}(q) \leftarrow 0$; $T^{out} \leftarrow T^{out} \cup \{\langle q, nop, q \rangle\}$
  else for *each* $t \in T$ *with* $t(\lambda(q)) \neq \bot$ **do**
    if $\exists q'.(q', q) \in (T^{out})^* . \lambda(q') = t(\lambda(q))$ **then**
      $T^{out} \leftarrow T^{out} \cup \{\langle q, \texttt{op}(t), q' \rangle\}$; $\lambda(\langle q, \texttt{op}(t), q' \rangle) \leftarrow t$
    else
      $\texttt{new}(q')$; $\kappa(q') \leftarrow \kappa(t(\lambda(q)))$; $\lambda(q') \leftarrow t(\lambda(q))$;
      if $\lambda(q') \in \Gamma_0$ **then** $Q_0^{out} \leftarrow Q_0^{out} \cup \{q'\}$ **else** $Q_1^{out} \leftarrow Q_1^{out} \cup \{q'\}$;
      $T^{out} \leftarrow T^{out} \cup \{\langle q, \texttt{op}(t), q' \rangle\}$; $\lambda(\langle q, \texttt{op}(t), q' \rangle) \leftarrow t$;
      $\texttt{ToExplore} \leftarrow \texttt{ToExplore} \cup \{q'\}$;

Operation of the algorithm. The algorithm performs a forward analysis similar to the classical Karp-Miller algorithm for Petri nets. We start with a given nonnegative partial configuration, explore its successors, create loops when previously visited configurations are repeated and define a special operation for the case when configurations strictly increase. The algorithm builds the graph of the game $\mathcal{G}^{out}$ successively (i.e., the set of states Q^{out}, the set of transitions T^{out}, and the coloring of states κ). Additionally, for bookkeeping purposes inside the algorithm and for reasoning about the correctness of the algorithm, we define a labeling function λ on the set of states and transitions in $\mathcal{G}^{out}$ such that each state in $\mathcal{G}^{out}$ is labeled by a nonnegative partial configuration in Γ^C, and each transition in $\mathcal{G}^{out}$ is labeled by a transition in $\mathcal{G}$.

The algorithm first computes the Pareto frontier $\texttt{Pareto}[\mathcal{G},\mathcal{V},0,C'](\texttt{Parity})$ for all counter sets $C' \subseteq \mathcal{C}$ with $|C'| = |C| - 1$. This is possible by the induction hypothesis. It stores the union of all these sets in β (line 1). At line 2, the algorithms initializes the set of transitions T^{out} to be empty, creates the first state q^{out}, defines its coloring to be the same as that of the state of the input nonnegative partial configuration γ, labels it by γ, and then adds it to the set of states Q^{out}. At line 3 it adds q^{out} to the set of states of Player 0 or Player 1 (depending on where γ belongs), and at line 4 it adds q^{out} to the set `ToExplore`. The latter contains the set of states that have been created but not yet analyzed by the algorithm.

After the initialization phase, the algorithm starts iterating the *while*-loop starting at line 5. During each iteration, it picks and removes a new state q from the set `ToExplore` (line 6). First, it checks two special conditions under which the game is made immediately losing (resp. winning) for Player 0.

Condition 1: If $\neg(\beta \lhd \lambda(q))$ (line 7), then we know by Lemma 7 (item 3) that the nonnegative partial configuration $\lambda(q)$ is not winning for Player 0 in $\mathcal{G}$.

Therefore, we make the state q losing for Player 0 in $\mathcal{G}^{out}$. To do that, we change the color of q to 1 (any odd color will do), and add a self-loop to q. Any continuation of a run from q is then losing for Player 0 in $\mathcal{G}^{out}$.

Condition 2: If Condition 1 did not hold then the algorithm checks (at line 9) whether there is a *predecessor* q' of q in $\mathcal{G}^{out}$ with a label $\lambda(q')$ that is *strictly* smaller than the label $\lambda(q)$ of q, i.e., $\lambda(q') \prec \lambda(q)$. (Note that we are not comparing q to arbitrary other states in $\mathcal{G}^{out}$, but only to predecessors.) If that is the case, then the state q is made winning for Player 0 in $\mathcal{G}^{out}$. To do that, we change the color of q to 0 (any even color will do), and add a self-loop to q. The intuition for making q winning for Player 0 is as follows. Since $\lambda(q') \prec \lambda(q)$, the path from $\lambda(q')$ to $\lambda(q)$ increases the value of at least one of the defined counters (those in C), and will not decrease the other counters in C (though it might have a negative effect on the undefined counters in $\mathcal{C} - C$). Thus, if a run in $\mathcal{G}$ iterates this path sufficiently many times, the value of at least one counter in C will be pumped and becomes sufficiently high to allow Player 0 to win the parity game on $\mathcal{G}$, provided that the counters in $\mathcal{C} - C$ are initially instantiated with sufficiently high values. This follows from the property $\beta \lhd \lambda(q)$ and Lemma 7 (items 1 and 2).

If none of the tests for Condition1/Condition2 at lines 7 and 9 succeeds, the algorithm continues expanding the graph of $\mathcal{G}^{out}$ from q. It generates all successors of q by applying each transition $t \in T$ in $\mathcal{G}$ to the label $\lambda(q)$ of q (line 11). If the result $t(\lambda(q))$

is defined then there are two possible cases. The first case occurs if we have previously encountered (and added to Q^{out}) a state q' whose label equals $t(\lambda(q))$ (line 12). Then we add a transition from q back to q' in $\mathcal{G}^{out}$, where the operation of the new transition is the same operation as that of t, and define the label of the new transition to be t. Otherwise (line 15), we create a new state q', label it with the nonnegative configuration $t(\lambda(q))$ and assign it the same color as $t(\lambda(q))$. At line 16 q^{out} is added to the set of states of Player 0 or Player 1 (depending on where γ belongs). We add a new transition between q and q' with the same operation as t. The new transition is labeled with t. Finally, we add the new state q' to the set of states to be explored.

Lemma 8. *Algorithm 1 will always terminate.*

Lemma 8 implies that the integer game $\mathcal{G}^{out}$ is finite (and hence well-defined). The following lemma shows the relation between the input and output games $\mathcal{G}, \mathcal{G}^{out}$.

Lemma 9. $[0, \mathcal{V}] : \gamma \models_{\mathcal{G}} \texttt{Parity}$ *iff* $[0, \mathcal{E}] : \gamma^{out} \models_{\mathcal{G}^{out}} \texttt{Parity} \wedge \Box\overline{\texttt{neg}}$.

Proof sketch. The left to right implication is easy. Given a Player 0 winning strategy in $\mathcal{G}$, one can construct a winning strategy in $\mathcal{G}^{out}$ that uses the same transitions, modulo the labeling function $\lambda()$. The condition $\Box\overline{\texttt{neg}}$ in $\mathcal{G}^{out}$ is satisfied since the configurations in $\mathcal{G}$ are always nonnegative and the parity condition is satisfied since the colors seen in corresponding plays in $\mathcal{G}^{out}$ and $\mathcal{G}$ are the same.

For the right to left implication we consider a Player 0 winning strategy σ_0 in $\mathcal{G}^{out}$ and construct a winning strategy σ_0' in $\mathcal{G}$. The idea is that a play π in $\mathcal{G}$ induces a play π' in $\mathcal{G}^{out}$ by using the same sequence of transitions, but removing all so-called *pumping sequences*, which are subsequences that end in **Condition 2**. Then σ_0' acts on history π like σ_0 on history π'. For a play according to σ_0' there are two cases. Either it will eventually reach a configuration that is sufficiently large (relative to β) such that a winning strategy is known by induction hypothesis. Otherwise it contains only finitely many pumping sequences and an infinite suffix of it coincides with an infinite suffix of a play according to σ_0 in $\mathcal{G}^{out}$. Thus it sees the same colors and satisfies $\texttt{Parity}$. □

Since γ^{out} is abstract and the abstract energy parity problem is decidable (Theorem 1) we obtain Theorem 3.

The termination proof in Lemma 8 relies on Dickson's Lemma, and thus there is no elementary upper bound on the complexity of Algorithm 1 or on the size of the constructed energy game $\mathcal{G}^{out}$. The algorithm in [4] for the fixed initial credit problem in pure energy games without the parity condition runs in d-exponential time (resp. $(d-1)$-exponential time for offsets encoded in unary) for dimension d, and is thus not elementary either. As noted in [4], the best known lower bound is EXPSPACE hardness, easily obtained via a reduction from the control-state reachability (i.e., coverability) problem for Petri nets.

5 Applications to Other Problems

5.1 Weak Simulation Preorder between VASS and Finite-State Systems

Weak simulation preorder [9] is a semantic preorder on the states of labeled transition graphs, which can be characterized by weak simulation games. A configuration of the

game is given by a pair of states (q_1, q_0). In every round of the game, Player 1 chooses a labeled step $q_1 \xrightarrow{a} q_1'$ for some label a. Then Player 0 must respond by a move which is either of the form $q_0 \xrightarrow{\tau^* a \tau^*} q_0'$ if $a \neq \tau$, or of the form $q_0 \xrightarrow{\tau^*} q_0'$ if $a = \tau$ (the special label τ is used to model internal transitions). The game continues from configuration (q_1', q_0'). A player wins if the other player cannot move and Player 0 wins every infinite play. One says that q_0 *weakly simulates* q_1 iff Player 0 has a winning strategy in the weak simulation game from (q_1, q_0). States in different transition systems can be compared by putting them side-by-side and considering them as a single transition system.

We use $\langle Q, T, \Sigma, \lambda \rangle$ to denote a labeled VASS where the states and transitions are defined as in Section 2, Σ is a finite set of labels and $\lambda : T \mapsto \Sigma$ assigns labels to transitions.

It was shown in [2] that it is decidable whether a finite-state labeled transition system weakly simulates a labeled VASS. However, the decidability of the reverse direction was open. (The problem is that the weak $\xrightarrow{\tau^* a \tau^*}$ moves in the VASS make the weak simulation game infinitely branching.) We now show that it is also decidable whether a labeled VASS weakly simulates a finite-state labeled transition system. In particular this implies that weak simulation equivalence between a labeled VASS and a finite-state labeled transition system is decidable. This is in contrast to the undecidability of weak *bisimulation* equivalence between VASS and finite-state systems [11].

Theorem 5. *It is decidable whether a labeled* VASS *weakly simulates a finite-state labeled transition system.*

Proof sketch. Given a labeled VASS and a finite-state labeled transition system, one constructs a single-sided VASS parity game s.t. the VASS weakly simulates the finite system iff Player 0 wins the parity game. The idea is to take a controlled product of the finite system and the VASS s.t. every round of the weak simulation game is encoded by a single move of Player 1 followed by an arbitrarily long sequence of moves by Player 0. The move of Player 1 does not change the counters, since it encodes a move in the finite system, and thus the game is single-sided. Moreover, one enforces that every sequence of consecutive moves by Player 0 is finite (though it can be arbitrarily long), by assigning an odd color to Player 0 states and a higher even color to Player 1 states.

5.2 μ-Calculus Model Checking VASS

While model checking VASS with linear-time temporal logics (like LTL and linear-time μ-calculus) is decidable [8,10], model checking VASS with most branching-time logics (like EF, EG, CTL and the modal μ-calculus) is undecidable [8]. However, we show that Theorem 3 yields the decidability of model checking single-sided VASS with a guarded fragment of the modal μ-calculus. We consider a VASS $\langle Q, T \rangle$ where the states, transitions and semantics are defined as in Section 2, and reuse the notion of partial configurations and the transition relation defined for the VASS semantics on integer games. We specify properties on such VASS in the positive μ-calculus L_μ^{pos} whose atomic propositions q refer to control-states $q \in Q$ of the input VASS.

The syntax of the positive μ-calculus L_μ^{pos} is given by the following grammar: $\phi ::= q \mid X \mid \phi \wedge \phi \mid \phi \vee \phi \mid \Diamond\phi \mid \Box\phi \mid \mu X.\phi \mid \nu X.\phi$ where $q \in Q$ and X belongs to a countable set of variables $\mathcal{X}$. The semantics of L_μ^{pos} is defined as usual (see [3]). To

each closed formula ϕ in L_μ^{pos} (i.e., without free variables) it assigns a subset of concrete configurations $[\![\phi]\!]$.

The model-checking problem of VASS with L_μ^{pos} can then be defined as follows. Given a VASS $\mathcal{S} = \langle Q, T \rangle$, a closed formula ϕ of L_μ^{pos} and an initial configuration γ_0 of $\mathcal{S}$, do we have $\gamma_0 \in [\![\phi]\!]$? If the answer is yes, we will write $\mathcal{S}, \gamma_0 \models \phi$. The more general *global model-checking problem* is to compute the set $[\![\phi]\!]$ of configurations that satisfy the formula. The general unrestricted version of this problem is undecidable.

Theorem 6. *[8] The model-checking problem of* VASS *with* L_μ^{pos} *is undecidable.*

One way to solve the μ-calculus model-checking problem for a given Kripke structure is to encode the problem into a parity game [12]. The idea is to construct a parity game whose states are pairs, where the first component is a state of the structure and the second component is a subformula of the given μ-calculus formula. States of the form $\langle q, \Box\phi \rangle$ or $\langle q, \phi \wedge \psi \rangle$ belong to Player 1 and the remainder belong to Player 0. The colors are assigned to reflect the nesting of least and greatest fixpoints. We can adapt this construction to our context by building an integer game from a formula in L_μ^{pos} and a VASS $\mathcal{S}$, as stated by the next lemma.

Lemma 10. *Let* $\mathcal{S}$ *be a* VASS, γ_0 *a concrete configuration of* $\mathcal{S}$ *and* ϕ *a closed formula in* L_μ^{pos}. *One can construct an integer game* $\mathcal{G}(\mathcal{S}, \phi)$ *and an initial concrete configuration* γ_0' *such that* $[0, \mathcal{V}] : \gamma_0' \models_{\mathcal{G}(\mathcal{S},\phi)} \texttt{Parity}$ *if and only if* $\mathcal{S}, \gamma_0 \models \phi$.

Now we show that, under certain restrictions on the considered VASS and on the formula from L_μ^{pos}, the constructed integer game $\mathcal{G}(\mathcal{S}, \phi)$ is single-sided, and hence we obtain the decidability of the model-checking problem from Theorem 3. First, we reuse the notion of single-sided games from Section 2 in the context of VASS, by saying that a VASS $\mathcal{S} = \langle Q, T \rangle$ is single-sided iff there is a partition of the set of states Q into two sets Q_0 and Q_1 such that $op = nop$ for all transitions $t \in T$ with $\texttt{source}(t) \in Q_1$. The guarded fragment L_μ^{sv} of L_μ^{pos} for single-sided VASS is then defined by guarding the $\Box$ operator with a predicate that enforces control-states in Q_1. Formally, the syntax of L_μ^{sv} is given by the following grammar: $\phi ::= q \mid X \mid \phi \wedge \phi \mid \phi \vee \phi \mid \Diamond\phi \mid Q_1 \wedge \Box\phi \mid \mu X.\phi \mid \nu X.\phi$, where Q_1 stands for the formula $\bigvee_{q \in Q_1} q$. By analyzing the construction of Lemma 10 in this restricted case, we obtain the following lemma.

Lemma 11. *If* $\mathcal{S}$ *is a single-sided* VASS *and* $\phi \in L_\mu^{sv}$ *then the game* $\mathcal{G}(\mathcal{S}, \phi)$ *is equivalent to a single-sided game.*

By combining the results of the last two lemmas with Corollary 1, Theorem 3 and Corollary 2, we get the following result on model checking single-sided VASS.

Theorem 7.

1. *Model checking* L_μ^{sv} *over single-sided* VASS *is decidable.*
2. *If* $\mathcal{S}$ *is a single-sided* VASS *and* ϕ *is a formula of* L_μ^{sv} *then* $[\![\phi]\!]$ *is upward-closed and its set of minimal elements is computable.*

6 Conclusion and Outlook

We have established a connection between multidimensional energy games and single-sided VASS games. Thus our algorithm to compute winning sets in VASS parity games can also be used to compute the minimal initial credit needed to win multidimensional energy parity games, i.e., the Pareto frontier.

It is possible to extend our results to integer parity games with a mixed semantics, where a subset of the counters follow the energy semantics and the rest follow the VASS semantics. If such a mixed parity game is single-sided w.r.t. the VASS counters (but not necessarily w.r.t. the energy counters) then it can be reduced to a single-sided VASS parity game by our construction in Section 3. The winning set of the derived single-sided VASS parity game can then be computed with the algorithm in Section 4.

References

1. Abdulla, P.A., Bouajjani, A., d'Orso, J.: Deciding monotonic games. In: Baaz, M., Makowsky, J.A. (eds.) CSL 2003. LNCS, vol. 2803, pp. 1–14. Springer, Heidelberg (2003)
2. Abdulla, P.A., Čerāns, K., Jonsson, B., Tsay, Y.: General decidability theorems for infinite-state systems. In: LICS 1996, pp. 313–321. IEEE (1996)
3. Abdulla, P.A., Mayr, R., Sangnier, A., Sproston, J.: Solving parity games on integer vectors. Technical Report EDI-INF-RR-1417, Univ. of Edinburgh (2013)
4. Brázdil, T., Jančar, P., Kučera, A.: Reachability games on extended vector addition systems with states. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 478–489. Springer, Heidelberg (2010)
5. Chatterjee, K., Doyen, L.: Energy parity games. TCS 458, 49–60 (2012)
6. Chatterjee, K., Doyen, L., Henzinger, T., Raskin, J.-F.: Generalized mean-payoff and energy games. In: FSTTCS 2010. LIPIcs, LZI, vol. 8, pp. 505–516 (2010)
7. Chatterjee, K., Randour, M., Raskin, J.-F.: Strategy synthesis for multi-dimensional quantitative objectives. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 115–131. Springer, Heidelberg (2012)
8. Esparza, J., Nielsen, M.: Decibility issues for Petri nets - a survey. Journal of Informatik Processing and Cybernetics 30(3), 143–160 (1994)
9. van Glabbeek, R.J.: The linear time – branching time spectrum I; the semantics of concrete, sequential processes. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, ch. 1, pp. 3–99. Elsevier (2001)
10. Habermehl, P.: On the complexity of the linear-time mu-calculus for Petri-nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 102–116. Springer, Heidelberg (1997)
11. Jančar, P., Esparza, J., Moller, F.: Petri nets and regular processes. J. Comput. Syst. Sci. 59(3), 476–503 (1999)
12. Kirsten, D.: Alternating tree automata and parity games. In: Grädel, E., Thomas, W., Wilke, T. (eds.) Automata, Logics, and Infinite Games. LNCS, vol. 2500, pp. 153–167. Springer, Heidelberg (2002)
13. McNaughton, R.: Infinite games played on finite graphs. Annals of Pure and Applied Logic 65, 149–184 (1993)
14. Raskin, J.-F., Samuelides, M., Van Begin, L.: Games for counting abstractions. Electr. Notes Theor. Comput. Sci. 128(6), 69–85 (2005)
15. Valk, R., Jantzen, M.: The residue of vector sets with applications to decidability problems in Petri nets. Acta Inf. 21, 643–674 (1985)
16. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. TCS 200, 135–183 (1998)

Well-Structured Pushdown Systems

Xiaojuan Cai[1] and Mizuhito Ogawa[2]

[1] BASICS Lab, Shanghai Jiao Tong University, China
cxj@sjtu.edu.cn
[2] Japan Advanced Institute of Science and Technology, Japan
mizuhito@jaist.ac.jp

Abstract. *Pushdown systems* (PDSs) model single-thread recursive programs, and *well-structured transition systems* (WSTSs), such as *vector addition systems*, are useful to represent non-recursive multi-thread programs. Combining these two ideas, our goal is to investigate *well-structured pushdown systems* (WSPDSs), pushdown systems with well-quasi-ordered control states and stack alphabet.

This paper focuses on subclasses of WSPDSs, in which the coverability becomes decidable. We apply WSTS-like techniques on classical P-automata. A $Post^*$-automata (resp. Pre^*-automata) construction is combined with Karp-Miller acceleration (resp. ideal representation) to characterize the set of successors (resp. predecessors) of given configurations. As examples, we show that the coverability is decidable for *recursive vector addition system with states*, *multi-set pushdown systems*, and a WSPDS with finite control states and well-quasi-ordered stack alphabet.

1 Introduction

There are two directions of infinite (discrete) state systems. A *pushdown system* (PDS) consists of finite control states and finite stack alphabet, where a stack stores the context. It is often used to models single-thread recursive programs.A *well-structured transition system* (WSTS) [1,10] consists of a well-quasi-ordered set of states. A *vector addition system* (VAS, or Petri Net) is its typical example. It often works for modeling dynamic thread creation of multi-thread program [2]. Our naive motivation comes from what happens when we combine them as a general framework for modeling recursive multi-thread programs.

A 3-thread boolean-valued recursive program with synchronization is enough to encode *Post-correspondence-problem* [19]. Thus, its reachability is undecidable. There are several decidable subclasses, which are typically reduced to single stack PDSs with infinite control states and stack alphabet.

- *Restrict the number of context switching (bounded reachability):* Context-bounded concurrent pushdown systems [18], and their extensions with dynamic thread creation [2].
- *Restrict interleaves among stack operations:* Multi-set pushdown systems (Multi-set PDSs) to model multi-thread asynchronous programs [20,13], and Recursive Vector Addition System with States (RVASS) to model multi-thread programs with fork/join synchronizations [3].

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 121–136, 2013.

A popular decidable property of ordinary PDSs is the *configuration reachability*, i.e., whether a target configuration is reachable from an initial configuration. A P-automaton construction [9,4,7] is its classical technique such that a $Post^*$ automaton accepts the set of successors of an initial configuration, and a Pre^* automaton accepts the set of predecessors of a target configuration.

A popular decidable property of WSTSs is *coverability*, i.e., whether an initial configuration reaches to that covers a target configuration. There are forward and backward techniques. As the former, Karp-Miller acceleration [8] for VASs is well-known, which was generalized in [11,12]. As the latter, an ideal (i.e., an upward closed set) representation is immediate [1,10], though less efficient. Note that the reachability of WSTSs is not easy. For instance, the reachability of VASs stays decidable, but it requires deep insight on Presburger arithmetic [16,15].

Our ultimate goal is to study *well-structured pushdown systems* (WSPDSs), pushdown systems with well-quasi-ordered control states and stack alphabet. This paper focuses on subclasses of WSPDSs, in which the coverability becomes decidable. We apply WSTS-like techniques on classical P-automata. A $Post^*$-automata (resp. Pre^*-automata) construction is combined with Karp-Miller acceleration (resp. ideal representation) to characterize the set of successors (resp. predecessors) of given configurations. As examples, we show that the coverability is decidable for RVASSs, Multi-set PDSs, and a WSPDS with finite control states and WQO stack alphabet. The first one extends the decidability of the state reachability of RVASSs [3] to the coverability, and the second one relaxes finite stack alphabet of Multi-set PDSs [20,13] to being well-quasi-ordered.

Related Work

Combining PDSs and VASs is not new. Process rewrite system (PRS) [17] is a pioneer work on such combination. A PRS is a(n AC) ground term rewriting system, consisting of the sequential composition ".", the parallel composition "|| ", and finitely many constants, which can be regarded as a PDS with finite control states and vector stack alphabet. The decidability of the reachability between ground terms was shown based on the reachability of a VAS. However, a PRS is rather weak to model multi-thread programs, since it cannot describe vector additions between adjacent stack frames during push/pop operations.

An RVASS [3] allows vector additions during pop rules. The state reachability was shown by reducing an RVASS to a Branching VASS [21]. Our WSPDS extends it to the coverability. A more general framework is a WQO automaton [5], which is a WSTS with auxiliary storage (e.g., stacks and queues). Although in general undecidable, its coverability becomes decidable under the compatibility of *rank* functions with a WQO. A Multi-set PDS [13,20] is a such instance.

Our drawback is difficulty to estimate complexity, due to the nature of well-quasi-ordering. s For instance, the coverability of a Branching VAS (BVAS) is 2EXPTIME-complete [6], and accordingly RVASS will be. Lower bounds of various VAS are reported by reduction to fragments of first-order logic [14]. However, we cannot directly conclude such estimations.

2 Preliminaries

2.1 Well-Structured Transition System

A *quasi-order* $(D, \leq)$ is a reflexive transitive binary relation on D. An upward closure of $X \subseteq D$, denoted by $X^{\uparrow}$, is the set of elements in D larger than those in X, i.e., $X^{\uparrow} = \{d \in D \mid \exists x \in X.x \leq d\}$). A subset I is an *ideal* if $I = I^{\uparrow}$. Similarly, a downward closure of $X \subseteq D$ is denoted by $X^{\downarrow} = \{d \in D \mid \exists x \in X.x \geq d\}$. We denote the set of all ideals by $\mathcal{I}(D)$. A quasi-order $(D, \leq)$ is a *well-quasi-order* (WQO) if, for each infinite sequence $a_1, a_2, a_3, \cdots$ in D, there exist i, j with $i < j$ and $a_i \leq a_j$.

Definition 1. *A* well-structured transition system *(WSTS) is a triplet* $M = \langle (P, \preceq), \rightarrow \rangle$ *where* $(P, \preceq)$ *is a WQO, and* $\rightarrow$ $(\subseteq P \times P)$ *is monotonic, i.e., for each* $p_1, q_1, p_2 \in P$, $p_1 \rightarrow q_1$ *and* $p_1 \preceq p_2$ *imply that there exists* q_2 *with* $p_2 \rightarrow q_2 \ \wedge q_1 \preceq q_2$.

Given two states $p, q \in P$, the *coverability* problem is to determine whether there exists q' with $q' \succeq q$ and $p \rightarrow^* q'$.

Vector addition systems (VAS) (equivalently, Petri net) are WSTSs with $\mathbb{N}^k$ as the set of states and a subtraction followed by an addition as a transition rule. The reachability problem of VAS is decidable, but its proof is complex [16,15]. The coverability also attracts attentions and is implemented, such as in **Pep**. [1] *Karp-Miller acceleration* is an efficient technique for the coverability. If there is a descendant vector (wrt transitions) strictly larger than one of its ancestors on coordinates, values at these coordinates are accelerated to ω.

There is an alternative backward method to decide coverability for a general WSTS. Starting from an ideal $\{q\}^{\uparrow}$, where q is the target state to be covered, its predecessors are repeatedly computed. Note that, for a WSTS and an ideal $I(\subseteq P)$, the predecessor set $pre(I) = \{p \in P \mid \exists q \in I.p \rightarrow q\}$ is also an ideal from the monotonicity. Its termination is obtained by the following lemma.

Lemma 1. *[10]* $(D, \leq)$ *is a WQO, if, and only if, any infinite sequence* $I_0 \subseteq I_1 \subseteq I_2 \subseteq \cdots$ *in* $\mathcal{I}(D)$ *eventually stabilize.*

From now on, we denote $\mathbb{N}$ (resp. $\mathbb{Z}$) for the set of natural numbers (resp. integers), and $\mathbb{N}^k$ (resp. $\mathbb{Z}^k$) is the set of k-dimensional vectors over $\mathbb{N}$ (resp. $\mathbb{Z}$). As notational convention, $\boldsymbol{n}, \boldsymbol{m}$ are for vectors in $\mathbb{N}^k$, $\boldsymbol{z}, \boldsymbol{z}'$ are for vectors in $\mathbb{Z}^k$, $\widetilde{\boldsymbol{n}}, \widetilde{\boldsymbol{m}}$ are for sequences of vectors.

2.2 Pushdown System

We define a pushdown system (PDS) with extra rules, *simple-push* and *nonstandard-pop*. These rules do not appear in the standard definition since they are encoded into standard rules. For example, a non-standard pop rule

[1] `http://theoretica.informatik.uni-oldenburg.de/~pep/`

$(p, \alpha\beta \to q, \gamma)$ is split into $(p, \alpha \to p_\alpha, \epsilon)$ and $(p_\alpha, \beta \to q, \gamma)$ by adding a fresh state p_α. However, later we will consider a PDS with infinite stack alphabet, and this encoding may change the context. For instance, for a PDS with finite control states and infinite stack alphabet, this encoding may lead infinite control states.

Definition 2. *A* pushdown system (PDS) *is a triplet* $\langle P, \Gamma, \Delta \rangle$ *where*

- P *is a finite set of states,*
- Γ *is finite stack alphabet, and*
- $\Delta \subseteq P \times \Gamma^{\leq 2} \times P \times \Gamma^{\leq 2}$ *is a finite set of transitions, where* $(p, v, q, w) \in \Delta$ *is denoted by* $(p, v \to q, w)$.

We use $\alpha, \beta, \gamma, \cdots$ to range over Γ, and $w, v, \cdots$ over words in Γ^*. A *configuration* $\langle p, w \rangle$ is a pair of a state p and a stack content (word) w. As convention, we denote configurations by $c_1, c_2, \cdots$. One step transition $\hookrightarrow$ between configurations is defined as follows. $\hookrightarrow^*$ is the reflexive transitive closure of $\hookrightarrow$.

$$\text{inter}\frac{(p, \gamma \to p', \gamma') \in \Delta}{\langle p, \gamma w \rangle \hookrightarrow \langle p', \gamma' w \rangle} \quad \text{push}\frac{(p, \gamma \to p', \alpha\beta) \in \Delta}{\langle p, \gamma w \rangle \hookrightarrow \langle p', \alpha\beta w \rangle} \quad \text{pop}\frac{(p, \gamma \to p', \epsilon) \in \Delta}{\langle p, \gamma w \rangle \hookrightarrow \langle p', w \rangle}$$

$$\text{simple-push}\frac{(p, \epsilon \to p', \alpha) \in \Delta}{\langle p, w \rangle \hookrightarrow \langle p', \alpha w \rangle} \quad \text{nonstandard-pop}\frac{(p, \alpha\beta \to p', \gamma) \in \Delta}{\langle p, \alpha\beta w \rangle \hookrightarrow \langle p', \gamma w \rangle}$$

A PDS enjoys decidable *configuration reachability*, i.e., given configurations $\langle p, w \rangle$, $\langle q, v \rangle$ with $p, q \in P$ and $w, v \in \Gamma^*$, decide whether $\langle p, w \rangle \hookrightarrow^* \langle q, v \rangle$.

3 WSPDS and P-Automata Technique

3.1 P-Automaton

A P-automaton is an automaton that accepts the set of reachable configurations of a PDS. P-automata are classified into $Post^*$-automata and Pre^*-automata,

Definition 3. *Given a PDS* $M = \langle P, \Gamma, \Delta \rangle$*, a P-automaton* $\mathcal{A}$ *is a quadruplet* (S, Γ, ∇, F) *where*

- F *is the set of final states, and* $P \subseteq S \setminus F$*, and*
- $\nabla \subseteq S \times (\Gamma \cup \{\epsilon\}) \times S$.

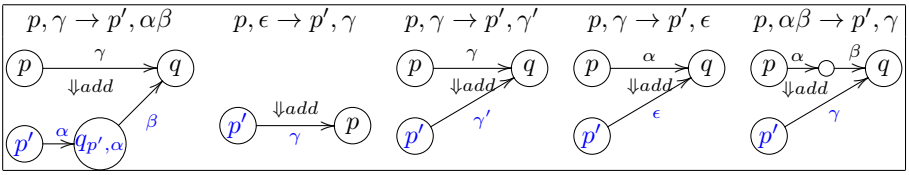
We write $s \overset{\gamma}{\mapsto} s'$ *for* $(s, \gamma, s') \in \nabla$ *and* $\Mapsto$ *for the reflexive transitive closure of* $\mapsto$*; It* accepts $\langle p, w \rangle$ *for* $p \in P$ *and* $w \in \Gamma^*$ *if* $p \overset{w}{\Longmapsto} f \in F$*. We use* $L(\mathcal{A})$ *to denote the set of configurations that* $\mathcal{A}$ *accepts. We assume that an* initial P-automaton *has no transitions* $s \overset{\gamma}{\mapsto} s'$ *with* $s' \in P$.

Let C_0 be a regular set of configurations of a PDS, and let $\mathcal{A}_0$ be an initial P-automaton that accepts C_0. The procedure to compute $post^*(C_0)$ starts from $\mathcal{A}_0$, and repeatedly adds edges according to the rules of a PDS until convergence. We call this procedure *saturation*. $Post^*$-saturation rules are given in Definition 4, which are illustrated in the following figure.

Definition 4. *For a PDS $\langle P, \Gamma, \Delta \rangle$, let $\mathcal{A}_0$ be an initial P-automaton accepting C_0. $Post^*(\mathcal{A}_0)$ is constructed by repeated applications of the following $Post^*$-saturation rules.*

$$\frac{(S, \Gamma, \nabla, F), \quad (p \overset{w}{\Longmapsto} q) \in \nabla}{(S \cup \{p'\}, \Gamma, \nabla \cup \{p' \overset{\gamma}{\mapsto} q\}, F)} \quad (p, w \to p', \gamma) \in \Delta, |w| \leq 2$$

$$\frac{(S, \Gamma, \nabla, F), \quad (p \overset{\gamma}{\mapsto} q) \in \nabla}{(S \cup \{p', q_{p',\alpha}\}, \Gamma, \nabla \cup \{p' \overset{\alpha}{\mapsto} q_{p',\alpha} \overset{\beta}{\mapsto} q\}, F)} \quad (p, \gamma \to p', \alpha\beta) \in \Delta$$



For instance, consider a push rule $(p, \gamma \to p', \alpha\beta)$. If $p \overset{\gamma}{\mapsto} q$ is in ∇, then $p' \overset{\alpha}{\mapsto} q_{p',\alpha} \overset{\beta}{\mapsto} q$ is added to ∇. The intuition is, if, for $v \in \Gamma^*$, $\langle p, \gamma v \rangle$ is in $post^*(C_0)$, then $\langle p', \alpha\beta v \rangle$ is also in $post^*(C_0)$ by applying rule $(p, \gamma \to p', \alpha\beta)$. The Pre^*-saturation rules to construct $pre^*(C_0)$ are similar, but in the reversal.

Remark 1. $Post^*$- (resp. Pre^*-) saturation introduces ϵ-transitions when applying standard pop rules (resp. simple push rules). ϵ-transitions make arguments complicated, and we assume preprocessing on PDSs.

1. The bottom symbol $\perp$ of the stack is explicitly prepared in Γ.
2. For $Post^*$-saturation, each standard pop rule $p, \alpha \to q, \epsilon$ is replaced with $(p, \alpha\gamma \to q, \gamma)$ for each $\gamma \in \Gamma$.
3. For Pre^*-saturation, each *simple push* rule $p, \epsilon \to q, \alpha$ is replaced with $(p, \gamma \to q, \alpha\gamma)$ for each $\gamma \in \Gamma$.

Lemma 2. *Let $\langle P, \Gamma, \Delta \rangle$ be a PDS, and let $\mathcal{A}_0$ be an initial P-automaton accepting C_0. Assume that $p \overset{w}{\Longmapsto} q$ in $Post^*(\mathcal{A}_0)$ and $p \in P$.*

1. *If $q \in P$, $\langle q, \epsilon \rangle \hookrightarrow^* \langle p, w \rangle$;*
2. *If $q \in S(\mathcal{A}_0) \backslash P$, there exists $q' \overset{v}{\Longmapsto} q$ in $\mathcal{A}_0$ with $q' \in P$ and $\langle q', v \rangle \hookrightarrow^* \langle p, w \rangle$.*

Its proof is a folklore (also in [23]). Lemma 2 shows that each accepted configuration is in $post^*(C_0)$ during the saturation process (*soundness*). On the other hand, $Post^*$ saturation rules put immediate successor configurations, and all configurations in $post^*(C_0)$ are finally accepted by $Post^*(\mathcal{A}_0)$ (*completeness*).

Theorem 1. *$post^*(C_0) = L(Post^*(\mathcal{A}_0))$, and $pre^*(C_0) = L(Pre^*(\mathcal{A}_0))$.*

For an ordinary PDS (i.e., with finite control states and stack alphabet), $Post^*(\mathcal{A}_0)$ and $Pre^*(\mathcal{A}_0)$ have bounded numbers of states. (Recall that each newly added state $q_{p,\gamma}$ has an index of a pair of a state and a stack symbol.)

Thus, the saturation procedure finitely converges. For a PDS with infinite control states and stack alphabet, although $Post^*(\mathcal{A}_0)$ and $Pre^*(\mathcal{A}_0)$ may not finitely converge, they converge as limits (of set unions). The same statement to Theorem 1 holds by Lemma 2' (a generalized Lemma 2) in [23]. In later sections (Section 4 and 5), we show when and how the finite convergence holds.

3.2 P-Automata for Coverability

We denote the set of partial functions from X to Y by $\mathcal{P}Fun(X, Y)$. Let $\underline{\ll}$, the quasi-ordering[2] on Γ^*, be the element-wise extension of $\leq$ on Γ, i.e., $\alpha_1 \cdots \alpha_n \underline{\ll} \beta_1 \cdots \beta_m$ if and only if $m = n$ and $\alpha_i \leq \beta_i$ for each i.

Definition 5. *A* well-structured pushdown system *(WSPDS) is a triplet* $M = \langle (P, \preceq), (\Gamma, \leq), \Delta \rangle$ *where*

- $(P, \preceq)$ *and* $(\Gamma, \leq)$ *are WQOs, and*
- $\Delta \subseteq \mathcal{P}Fun(P, P) \times \mathcal{P}Fun(\Gamma^{\leq 2}, \Gamma^{\leq 2})$ *is the finite set of monotonic transitions rules (wrt* $\preceq$ *and* $\underline{\ll}$*). We denote* $(p, w \to \phi(p), \psi(w))$ *if* $(\phi, \psi) \in \Delta$*,* $p \in Dom(\phi)$*, and* $w \in Dome(\psi)$ *hold.*

A PDS is a WSPDS with finite P and finite Γ, and WSTS is a WSPDS with a single control state and internal transition rules only (i.e., no push/pop rules). Note that $Dom(\psi)$ and $Dome(\phi)$ are upward-closed sets from their monotonicity. Instead of reachability, we consider the *coverability* on WSPDSs.

- **Coverability**: Given configurations $\langle p, w \rangle$, $\langle q, v \rangle$ with $p, q \in P$ and $w, v \in \Gamma^*$, we say $\langle p, w \rangle$ *covers* $\langle q, v \rangle$ if there exist $q' \succeq q$ and $v' \underline{\gg} v$ s.t. $\langle p, w \rangle \hookrightarrow^* \langle q', v' \rangle$. Coverability problem is to decide whether $\langle p, w \rangle$ *covers* $\langle q, v \rangle$.

Remark 2. Thanks to an anonymous referee, the coverability of a WSPDS is reduced to the state reachability. Let $v = \alpha_n \cdots \alpha_1 \bot$ and $v' = \beta_n \cdots \beta_1 \bot$. For fresh states $q_n, \cdots, q_1, q_0$ (incomparable wrt $\preceq$), add transition rules

$$\{(q', x \to q_n, \epsilon) \text{ if } x \geq \alpha_n \text{ and } q' \succeq q, \quad (q_{i+1}, x \to q_i, \epsilon) \text{ if } x \geq \alpha_i, \quad (q_1, \bot \to q_0, \bot)\}.$$

Then, the coverability (from $\langle p, w \rangle$ to $\langle q, v \rangle$) is reduced to the state reachability (from $\langle p, w \rangle$ to q_0). Note that the same technique (replacing $\geq$ and $\succeq$ with $=$) does not work for the configuration reachability, since it violates the monotonicity. Nevertheless, we keep focusing on the coverability, since

- Transition rules above are not permitted as an RVASS and a Multi-set PDS. Thus, the coverability is still more than the state reachability at the level of RVASSs and Multi-set PDSs.
- Proofs are mostly by induction on the saturation steps of P-automata construction. The coverability fits for describing their inductive invariants.

[2] In general, $\underline{\ll}$ is not a well-quasi-ordering, even if $\leq$ is.

There are two ways to decide the coverability. The forward method starts from an initial configuration $\langle p, w\rangle$, and computes the downward closure of its successor configurations. The backward method starts from a target configuration $\langle q, v\rangle$, and computes the downward closure of its predecessor configurations.

- **(Post)** $\mathcal{A}$ accepts the downward closure of successors of C_0, i.e., $L(\mathcal{A}) = \bigcup_{i\geq 0}(post^i(C_0)^{\downarrow}) = (\bigcup_{i\geq 0} post^i(C_0))^{\downarrow} = (post^*(C_0))^{\downarrow}$.
- **(Pre)** $\mathcal{A}$ accepts predecessors of the upward closure $C_0^{\uparrow}$ of C_0, i.e., $L(\mathcal{A}) = \bigcup_{i\geq 0} pre^i(C_0^{\uparrow}) = pre^*(C_0^{\uparrow})$.

Remark 3. As in Remark 1, we preprocess WSPDSs to eliminate standard pop rules for $Post^*$-saturation and simple push rules for Pre^*-saturation. In later decidability results on WSPDSs, the finiteness of transition rules is crucial. The following replacement keeps the monotonicity and the finiteness.

- In $Post^*$-saturation, a standard pop rule $\psi(\gamma) = \epsilon$ is replaced with $\psi'(\gamma\gamma') = \gamma'$.
- In Pre^*-saturation, a simple push rule $\psi(\epsilon) = \gamma$ is replaced with $\psi'(\gamma') = \gamma\gamma'$.

4 *Post**-automata for Coverability

Coverability is decidable if either $Post^*$ or Pre^*-saturation finitely converges. In this section, we consider a strictly monotonic WSPDS with finitely many control states, with $\mathbb{N}^k$ as stack alphabet, and without *standard push* rules. Such a PDS is a *Pushdown Vector Addition Systems*. Our choice comes from that $Post^*$-saturation for *standard push* rules introduce fresh states (which lead infinite exploration), and the strict monotonicity validates Karp-Miller acceleration.

We write $\mathbb{N}_\omega$ for $\mathbb{N}\cup\{\omega\}$. Let us fix the dimension $k > 0$ and let $j(\boldsymbol{n})$ be the j-th element of a vector $\boldsymbol{n} \in \mathbb{N}_\omega^k$. The zero-vector is denoted by $\mathbf{0}$ with $j(\mathbf{0}) = 0$ for each $j \leq k$. A sequence of vectors is denoted with a tilde, like $\widetilde{\boldsymbol{n}}$. For $J \subseteq [1..k]$, we define the following orderings on vectors:

- $\boldsymbol{n} <_J \boldsymbol{n}'$ if $j(\boldsymbol{n}) < j(\boldsymbol{n}')$ for $j \in J$ and $j(\boldsymbol{n}) = j(\boldsymbol{n}')$ for $j \notin J$.
- $\boldsymbol{n} \leq_J \boldsymbol{n}'$ if $j(\boldsymbol{n}) \leq j(\boldsymbol{n}')$ for $j \in J$ and $j(\boldsymbol{n}) = j(\boldsymbol{n}')$ for $j \notin J$.
- $\boldsymbol{n}_1 \cdots \boldsymbol{n}_l \underline{\ll}_J \boldsymbol{n}'_1 \cdots \boldsymbol{n}'_{l'}$ if $l = l'$ and $\boldsymbol{n}_i \leq_J \boldsymbol{n}'_i$ for each $i \leq l$.
- $\boldsymbol{n}_1 \cdots \boldsymbol{n}_l \ll_J \boldsymbol{n}'_1 \cdots \boldsymbol{n}'_{l'}$ if $\boldsymbol{n}_1 \cdots \boldsymbol{n}_l \underline{\ll}_J \boldsymbol{n}'_1 \cdots \boldsymbol{n}'_{l'}$ and $\boldsymbol{n}_i <_J \boldsymbol{n}'_i$ for some i.

For example, $(1,2) <_{\{2\}} (1,3)$, $(1,2) \leq_{\{1,2\}} (1,3)$, $(1,2)(1,1) \underline{\ll}_{\{1,2\}} (1,3)(1,1)$, and $(1,2)(1,1) \not\ll_{\{1,2\}} (1,3)(1,1)$. We will omit J of $\leq_J$ if $J = \{1..k\}$.

If $\boldsymbol{n} <_J \boldsymbol{n}'$, an *acceleration* $\boldsymbol{n} \upharpoonright \boldsymbol{n}'$ is given by $\boldsymbol{n}_J^{\uparrow}$ where $j(\boldsymbol{n}_J^{\uparrow}) = \omega$ if $j \in J$, and $j(\boldsymbol{n}_J^{\uparrow}) = j(\boldsymbol{n})$ otherwise. For example, $(1,2) \upharpoonright (2,2) = (1,2)_{\{1\}}^{\uparrow} = (\omega, 2)$.

Definition 6. *Fix $k \in \mathbb{N}$. A* Pushdown Vector Addition Systems *(PDVAS) is a WSPDS $\langle P, (\mathbb{N}^k, \leq), \Delta\rangle$ where*

- *P is finite.*
- *$\Delta \in P \times P \times \mathcal{P}Fun((\mathbb{N}^k)^{\leq 2}, \mathbb{N}^k)$ is finite and without standard push rules.*
- *ψ is effectively computed and* strictly monotonic *wrt $\ll_J$ for each rule $(p, q, \psi) \in \Delta$ and $J \subseteq [1..k]$.*

Strict monotonicity wrt $\ll_J$ is crucial for acceleration, which naturally holds in VASs. A VAS transition $\boldsymbol{n} \hookrightarrow \boldsymbol{n} + \boldsymbol{z}$ holds $\boldsymbol{n}' + \boldsymbol{z} >_J \boldsymbol{n} + \boldsymbol{z}$ for each $\boldsymbol{n}' >_J \boldsymbol{n}$. A WSPDS may have a non-standard pop rule $(p, \boldsymbol{n}_1\boldsymbol{n}_2 \to q, \boldsymbol{m})$, and we require that the growth of either $\boldsymbol{n}_1$ or $\boldsymbol{n}_2$ leads the growth of $\boldsymbol{m}$.

4.1 Dependency

Acceleration for a VAS occurs when a descendant is strictly larger than some of its ancestors. However, for a PDVAS, such descendant-ancestor relation is not obvious in a P-automaton. We introduce *dependency* $\Rrightarrow$ on P-automata transitions $\mapsto$. The dependency is generated during $Post^*$-saturation steps.

Definition 7. *For a PDS $\langle P, \Gamma, \Delta \rangle$, a* dependency $\Rrightarrow$ *over transitions of a $Post^*$-automaton is generated during the saturation procedure, starting from $\emptyset$.*

1. *If a transition $p' \overset{\beta}{\mapsto} q$ is added from a rule $(p, \alpha \to p', \beta)$ and transition $p \overset{\alpha}{\mapsto} q$, then $(p \overset{\alpha}{\mapsto} q) \Rrightarrow (p' \overset{\beta}{\mapsto} q)$.*
2. *If a transition $p' \overset{\gamma}{\mapsto} q$ is added from a rule $(p, \alpha\beta \to p', \gamma)$ and transitions $p \overset{\alpha}{\mapsto} q' \overset{\beta}{\mapsto} q$, then $(p \overset{\alpha}{\mapsto} q') \Rrightarrow (p' \overset{\gamma}{\mapsto} q)$ and $(q' \overset{\beta}{\mapsto} q) \Rrightarrow (p' \overset{\gamma}{\mapsto} q)$.*
3. *Otherwise, we do not update $\Rrightarrow$.*

We denote the reflexive transitive closure of $\Rrightarrow$ by $\Rrightarrow^*$. Strict monotonicity leads to the following lemma, which guarantees the soundness of accelerations.

Lemma 3. *For a $Post^*$-automaton $\mathcal{A}$ of a PDVAS, if $p \overset{\boldsymbol{n}}{\mapsto} q \Rrightarrow^* p' \overset{\boldsymbol{m}}{\mapsto} q'$ and $p \overset{\boldsymbol{n}'}{\mapsto} q \in \nabla(\mathcal{A})$ for $\boldsymbol{n}' >_J \boldsymbol{n}$ hold, there exists $\boldsymbol{m}' >_J \boldsymbol{m}$ such that $p' \overset{\boldsymbol{m}'}{\Longmapsto} q' \in \nabla(\mathcal{A})$ and $p \overset{\boldsymbol{n}'}{\mapsto} q \Rrightarrow^* p' \overset{\boldsymbol{m}'}{\Longmapsto} q'$.*

Note that, if $(p \overset{\boldsymbol{n}}{\mapsto} q) \Rrightarrow^* (p \overset{\boldsymbol{n}_1}{\mapsto} q)$ and $\boldsymbol{n} <_J \boldsymbol{n}_1$ hold, Lemma 3 concludes

$$(p \overset{\boldsymbol{n}}{\mapsto} q) \Rrightarrow^* (p \overset{\boldsymbol{n}_1}{\mapsto} q) \Rrightarrow^* (p \overset{\boldsymbol{n}_2}{\mapsto} q) \Rrightarrow^* \cdots \Rrightarrow^* (p \overset{\boldsymbol{n}_i}{\mapsto} q) \Rrightarrow^* \cdots$$

with $\boldsymbol{n}_i <_J \boldsymbol{n}_{i+1}$ for each i. Thus, we can safely apply the acceleration on J.

4.2 $Post^*_F$-saturation

As in Section 4.1, accelerations will occur when $p \overset{\boldsymbol{n}}{\mapsto} q \Rrightarrow^* p \overset{\boldsymbol{n}'}{\mapsto} q$ and $\boldsymbol{n} <_J \boldsymbol{n}'$ is found for some p, q and J during the $Post^*$-saturation steps. We combine dependency generation and accelerations into the post saturation rules for a PDVAS. This new saturation procedure is denoted by $Post^*_F$, and a resulting P-automaton is called a $Post^*_F$-automaton.

We conservatively extend ψ in a PDVAS, from $(\mathbb{N}^k)^{\leq 2} \to \mathbb{N}^k$ to $(\mathbb{N}^k_\omega)^{\leq 2} \to \mathbb{N}^k_\omega$ by $\psi(\widetilde{\boldsymbol{n}}) = sup\{\psi(\widetilde{\boldsymbol{n}}') \mid \widetilde{\boldsymbol{n}}' \in (\mathbb{N}^k)^{\leq 2}, \widetilde{\boldsymbol{n}}' \underline{\ll} \widetilde{\boldsymbol{n}}\}$ for $\widetilde{\boldsymbol{n}} \in (\mathbb{N}^k_\omega)^{\leq 2}$,

Definition 8. *For a PDVAS* $\langle P, (\mathbb{N}^k, \leq), \Delta \rangle$*, let* $\mathcal{A}_0 = (S_0, (\mathbb{N}^k_\omega, \leq), (\nabla_0, \emptyset), F)$ *be an initial P-automaton accepting* C_0*.* $Post^*_F(\mathcal{A}_0)$ *is the result of repeated applications of the following* $Post^*_F$ *saturation rules.*

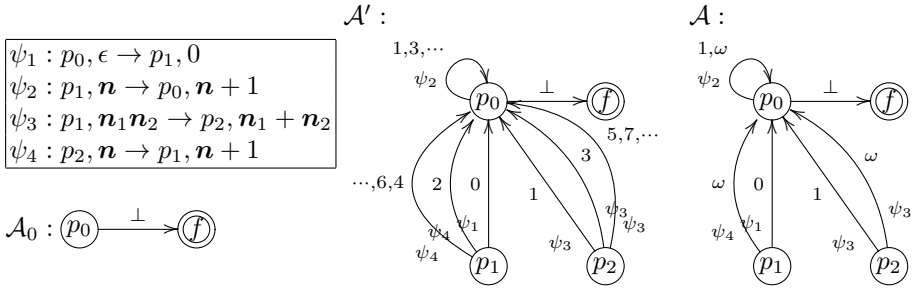
$$\frac{(S, \Gamma, (\nabla, \Rightarrow), F), \quad p \overset{\widetilde{\boldsymbol{n}}}{\Longmapsto} q}{(S \cup \{p'\}, \Gamma, (\nabla, \Rightarrow) \oplus (p' \overset{\boldsymbol{n}}{\mapsto} q, \Rightarrow'), F)} \quad (p, p', \psi) \in \Delta,\ \psi(\widetilde{\boldsymbol{n}}) = \boldsymbol{n}$$

where $\Rightarrow'$ *is the dependency newly added by Definition 7.*[3] *The operation* $\oplus$ *is defined as* $(\nabla, \Rightarrow) \oplus (p' \overset{\boldsymbol{n}}{\mapsto} q, \Rightarrow') =$

$$\begin{cases} (\nabla \cup \{p' \overset{\boldsymbol{n}' \restriction \boldsymbol{n}}{\longmapsto} q\}, \Rightarrow \cup \Rightarrow'_{\restriction}) & \text{if there exists } p' \overset{\boldsymbol{n}'}{\mapsto} q \in \nabla \text{ such that} \\ & p' \overset{\boldsymbol{n}'}{\mapsto} q \Rightarrow^* \cdot \Rightarrow' p' \overset{\boldsymbol{n}}{\mapsto} q \text{ and } \boldsymbol{n}' <_J \boldsymbol{n} \text{ for } J \neq \phi \\ (\nabla \cup \{p' \overset{\boldsymbol{n}}{\mapsto} q\}, \Rightarrow \cup \Rightarrow') & \text{otherwise} \end{cases}$$

where $\Rightarrow'_{\restriction}$ *is obtained from* $\Rightarrow'$ *by replacing its destination* $p' \overset{\boldsymbol{n}}{\mapsto} q$ *with* $p' \overset{\boldsymbol{n}' \restriction \boldsymbol{n}}{\longmapsto} q$*.*

Example 1. The following figure shows a $Post^*$-automaton $\mathcal{A}'$ and a $Post^*_F$-automaton $\mathcal{A}$ of a PDVAS with transition rules $\psi_1, \psi_2, \psi_3, \psi_4$. An initial configuration $C_0 = \{\langle p_0, \bot \rangle\}$ is accepted by $\mathcal{A}_0$. In $\mathcal{A}'$, $p_2 \overset{1}{\mapsto} p_0$ is generated from $p_1 \overset{0}{\mapsto} p_0 \overset{1}{\mapsto} p_0$ by ψ_3, and $p_1 \overset{2}{\mapsto} p_0$ is generated from $p_2 \overset{1}{\mapsto} p_0$ by ψ_4. Similarly, infinitely many $p_1 \overset{2k}{\Longmapsto} p_0$'s (and others) are generated. In $\mathcal{A}$, we have $(p_1 \overset{0}{\mapsto} p_0) \Rightarrow (p_2 \overset{1}{\mapsto} p_0) \Rightarrow (p_1 \overset{2}{\mapsto} p_0)$. An acceleration adds $(p_1 \overset{\omega}{\mapsto} p_0)$ instead of $(p_1 \overset{2}{\mapsto} p_0)$. Then, $p_2 \overset{\omega}{\mapsto} p_0$ and $p_0 \overset{\omega}{\mapsto} p_0$ are added by ψ_3 and ψ_2, respectively. This shows finitely convergence to $\mathcal{A}$, and we obtain $(post^*(C_0))^\downarrow = L(\mathcal{A})^\downarrow \cap (\mathbb{N}^k)^*$.



An immediate observation is that each configuration in $L(Post^*(\mathcal{A}_0))$ is covered by some in $L(Post^*_F(\mathcal{A}_0))$. The opposite follows from Lemma 4, which says that the downward closure (in $\mathbb{N}^k$) of a transition in $Post^*_F(\mathcal{A}_0)$ is included in the downward closure of transitions in $Post^*(\mathcal{A}_0)$. Its proof is found in [23].

Lemma 4. *For a PDVAS, let* $\mathcal{A}_0$ *be an initial P-automaton. If* $p \overset{\boldsymbol{n}}{\mapsto} q$ *is in* $Post^*_F(\mathcal{A}_0)$*, for each* $\boldsymbol{n}' \leq \boldsymbol{n}$ *with* $\boldsymbol{n}' \in \mathbb{N}^k$*, there exists* $\boldsymbol{n}''$ *such that* $p \overset{\boldsymbol{n}''}{\Longmapsto} q$ *is in* $Post^*(\mathcal{A}_0)$ *and* $\boldsymbol{n}' \leq \boldsymbol{n}'' \leq \boldsymbol{n}$*.*

[3] $\Rightarrow' = \emptyset$ if (p, p', ψ) is a push rule; otherwise, the destination of $\Rightarrow'$ is $p' \overset{\boldsymbol{n}}{\mapsto} q$.

Since a PDVAS does not have standard-push rules, the saturation procedure does not add new states. Thus, the sets of states in $Post^*_F(\mathcal{A}_0)$ and $Post^*(\mathcal{A}_0)$ are the same. From Lemma 4, we can obtain $L(Post^*_F(\mathcal{A}_0))^{\downarrow} \cap (\mathbb{N}^k)^* = (post^*(C_0))^{\downarrow}$.

Finite convergence of $Post^*_F$-saturation follows from that $\{(p, \boldsymbol{n}, q) \mid p, q \in S, \boldsymbol{n} \in \mathbb{N}^k_\omega\}$ is well-quasi-ordered. Thus, since accelerations can occur only finitely many times on a path of $\Rrightarrow^*$, the length of $\Rrightarrow^*$ is finite. Since $\Rrightarrow^*$ is finitely branching, König's lemma concludes that the $\Rrightarrow$-tree is finite.

Theorem 2. *For a PDVAS, if an initial P-automaton $\mathcal{A}_0$ with $L(\mathcal{A}_0) = C_0$ is finite, $Post^*_F(\mathcal{A}_0)$ finitely converges with $L(Post^*_F(\mathcal{A}_0))^{\downarrow} \cap (\mathbb{N}^k)^* = (post^*(C_0))^{\downarrow}$.*

4.3 Coverability of RVASS

In this section, we show that Recursive Vector Addition Systems with States (RVASSs) [3] are special cases of PDVASs, and Theorem refthm:termination implies decidability of its coverability.

Definition 9. *[3] Fix $k \in \mathbb{N}$. An* RVASS $\langle Q, \delta\rangle$ *consists of finite sets $\mathcal{Q}$ and δ of states and transitions, respectively. We denote*

- $q \xrightarrow{\boldsymbol{z}} q'$ *if* $(q, q', \boldsymbol{z}) \in \delta$ *for* $\boldsymbol{z} \in \mathbb{Z}^k$, *and*
- $q \xrightarrow{q_1 q_2} q'$ *if* $(q, q_1, q_2, q') \in \delta$.

The configuration $c \in (Q \times \mathbb{N}^k)^*$ represents a stack of pairs $\langle p, \boldsymbol{n}\rangle$ where $p \in Q$ and $\boldsymbol{n} \in \mathbb{N}^k$. The semantics is defined by following rules:

$$\frac{q \xrightarrow{\boldsymbol{z}} q' \quad \boldsymbol{n} + \boldsymbol{z} \in \mathbb{N}^k}{\langle q, \boldsymbol{n}\rangle c \hookrightarrow \langle q', \boldsymbol{n} + \boldsymbol{z}\rangle c} \qquad \frac{q \xrightarrow{q_1 q_2} q'}{\langle q, \boldsymbol{n}\rangle c \hookrightarrow \langle q_1, \boldsymbol{0}\rangle\langle q, \boldsymbol{n}\rangle c} \qquad \frac{q \xrightarrow{q_1 q_2} q'}{\langle q_2, \boldsymbol{n}'\rangle\langle q, \boldsymbol{n}\rangle c \hookrightarrow \langle q', \boldsymbol{n} + \boldsymbol{n}'\rangle c}$$

The *state-reachability* problem of an RVASS is, given two states q_0, q_f, whether there exist a vector $\boldsymbol{n}$ and a configuration c such that $\langle q_0, \boldsymbol{0}\rangle \hookrightarrow^* \langle q_f, \boldsymbol{n}\rangle c$. Lemma 3 in [3] showed its decidability by a reduction to a Branching VASS [6]. Below, Corollary 1 shows the decidability of the coverability. Note that the state reachability is the coverability from $\langle q_0, \boldsymbol{0}\rangle$ to $\{\langle q_f, \boldsymbol{0}^{\uparrow}\rangle\ \mathtt{any}^*\}$.

The encoding from an RVASS to a PDVAS is straightforward by regarding a configuration of an RVASS as a stack content in a PDVAS with a single control state $\bullet$, where $\langle q_i, (n_1, \cdots, n_k)\rangle \in Q \times \mathbb{N}^k$ is regarded as an element in $\Gamma = \mathbb{N}^{|Q|k}$

$$(\underbrace{0, \cdots, 0}_{(i-1)k}, n_1, \cdots, n_k, \underbrace{0, \cdots, 0}_{(|Q| - i)k})$$

Definition 10. *For $k \in \mathbb{N}$ and an* RVASS $R = \langle Q, \delta\rangle$, *a PDVAS $M_R = (\{\bullet\}, \Gamma, \Delta)$ consists of $\Gamma = \mathbb{N}^{|Q|k}$ and $\Delta \subseteq \{\bullet\} \times \{\bullet\} \times \mathcal{P}Fun(\Gamma^{\leq 2}, \Gamma)$ with*

1. *if $(q, q', \boldsymbol{z}) \in \delta$, then $(\bullet, \langle q, \boldsymbol{n}\rangle \to \bullet, \langle q', \boldsymbol{n} + \boldsymbol{z}\rangle) \in \Delta$.*
2. *if $(q, q_1, q_2, q') \in \delta$, then*
   (a) $(\bullet, \langle q, \epsilon\rangle \to \bullet, \langle q_1, \boldsymbol{0}\rangle) \in \Delta$ and (b) $(\bullet, \langle q_2, \boldsymbol{n}\rangle\langle q, \boldsymbol{m}\rangle \to \bullet, \langle q', \boldsymbol{n}+\boldsymbol{m}\rangle) \in \Delta$.

Corollary 1. *The coverability of an RVASS is decidable.*

5 Pre^*-automata for Coverability

When Δ has no non-standard pop rules, Pre^* does not introduce any fresh states, and we will show that ideal representations leads finite convergence. In this section, we assume that Δ has no non-standard pop rules.

5.1 Ideal Representation of Pre^*-automata

As mentioned in Section 3.2, we need to construct a Pre^*-automaton that accepts predecessors of an ideal $C_0^{\uparrow}$. A naive representation of such upward closures may be infinite. Therefore, we use an ideal representation Pre_F^*-automaton in which transition labels and states are ideals. Thanks to WQO, an ideal is characterized by its finitely many minimal elements, and ideals are well founded wrt set inclusion.

Definition 11. *For a WSPDS $\langle (P, \preceq), (\Gamma, \leq), \Delta \rangle$, by replacing Γ with $\mathcal{I}(\Gamma)$ and $P \subseteq S \setminus F$ with $\mathcal{I}(P) \subseteq S \setminus F$ in Definition 3, we obtain the definition of a Pre_F^*-automaton $\mathcal{A} = (S, \mathcal{I}(\Gamma), \nabla, F)$.*

As notational convention, let s, t to range over S, ideals K, K' to range over $\mathcal{I}(P)$, and I, I' over $\mathcal{I}(\Gamma)$. We denote $w \in \widetilde{I}$ for $\widetilde{I} = I_1 I_2 \cdots I_n$, if $w = \alpha_1 \alpha_2 \cdots \alpha_n$ and $\alpha_i \in I_i$ for each i. We say that $\mathcal{A}$ accepts a configuration $\langle p, w \rangle$, if there is a path $K \overset{\widetilde{I}}{\Longmapsto} f \in F$ in $\mathcal{A}$ and $p \in K$, $w \in \widetilde{I}$. The ideal representation of an initial P-automaton accepting $C_0^{\uparrow}$ is obtained from a P-automaton accepting C_0 by replacing each state p with $\{p\}^{\uparrow}$ and each transition label α with $\{\alpha\}^{\uparrow}$.

Definition 12. *Let $\mathcal{A}_0$ be an initial Pre_F^*-automaton accepting $C_0^{\uparrow}$. $Pre_F^*(\mathcal{A}_0)$ is the result of repeated applications of the following Pre_F^*-saturation rules*

$$\frac{(S, \mathcal{I}(\Gamma), \nabla, F), \quad K \overset{\widetilde{I}}{\Longmapsto} s}{(S, \mathcal{I}(\Gamma), \nabla, F) \oplus \{\phi^{-1}(K) \overset{\psi^{-1}(\widetilde{I})}{\longmapsto} s\}} \text{ if } \widetilde{I} \in \mathcal{I}(\Gamma^{\leq 2}) \text{ and } (\phi, \psi) \in \Delta$$

where $\phi^{-1}(K) \neq \emptyset$, $\psi^{-1}(\widetilde{I}) \neq \emptyset$, and $(S, \Sigma, \nabla, F) \oplus \{K \overset{I}{\mapsto} s\}$ is

$$\begin{cases} (S, \Sigma, \nabla, F) & \text{if } (K' \overset{I'}{\mapsto} s) \in \nabla \text{ with } K \subseteq K' \text{ and } I \subseteq I' \\ (S, \Sigma, (\nabla \setminus \{K \overset{I'}{\mapsto} s\}) \cup \{K \overset{I' \cup I}{\longmapsto} s\}, F) & \text{if } (K \overset{I'}{\mapsto} s) \in \nabla \\ (S \cup \{K\}, \Sigma, \nabla \cup \{K \overset{I}{\mapsto} s\}, F) & \text{otherwise} \end{cases}$$

The $\oplus$ operator merges ideals associated to transitions. Assume that a new transition $K \overset{I}{\mapsto} s$ is generated. If there is a transition $K' \overset{I'}{\mapsto} s$ with the same s, $K \subseteq K'$, and $I \subseteq I'$, the ideal of configurations starting from $K \overset{I}{\mapsto} s$ is included in that from $K' \overset{I'}{\mapsto} s$. Thus, no needs to add it. If there is a transition $K \overset{I'}{\mapsto} s$ between the same pair K, s, then take the union $I \cup I'$. Otherwise, we add a new transition.

It is easy to see that if $\phi \in \mathcal{P}Fun(X, Y)$ is monotonic, then, for any $I \in \mathcal{I}(Y)$, $\phi^{-1}(I)$ is an ideal in $\mathcal{I}(X)$. Completeness $pre^*(C_0^{\uparrow}) \subseteq L(Pre_F^*(\mathcal{A}_0))$ follows immediately by induction on saturation steps. Soundness $pre^*(C_0^{\uparrow}) \supseteq L(pre^*(\mathcal{A}_0))$ is guaranteed by Lemma 5, which is an invariant during the saturation procedure.

Lemma 5. *Assume $K \overset{\widetilde{I}}{\Longmapsto} s$ in $Pre_F^*(\mathcal{A}_0)$. For each $p \in K$, $w \in \widetilde{I}$,*

- *if $s = K' \in \mathcal{I}(P)$, then $\langle p, w\rangle \hookrightarrow^* \langle q, \epsilon\rangle$ for some $q \in K'$.*
- *if $s \notin \mathcal{I}(P)$, there exists $K' \overset{\widetilde{I'}}{\Longmapsto} s$ in $\mathcal{A}_0$ such that $\langle p, w\rangle \hookrightarrow^* \langle p', w'\rangle$ for some $p' \in K'$ and $w' \in \widetilde{I'}$.*

Theorem 3. *For an initial P-automaton $\mathcal{A}_0$ accepting $C_0^{\uparrow}$, $L(Pre_F^*(\mathcal{A}_0)) = pre^*(C_0^{\uparrow})$.*

Note that Theorem 3 only shows the correctness of Pre_F^*-saturation. We do not assume its finite convergence, which will be discussed in next two subsections.

5.2 Coverability of Multi-set PDS

As an example of the finite convergence, we show *Multi-set pushdown system* (Multi-set PDS) proposed in [20,13], which is an extension of PDS by attaching a multi-set into the configuration. We directly give the definition of a Multi-set PDS as a WSPDS. Note that, although a Multi-set PDS has infinitely many control states, it finitely converges because of restrictions on decreasing rules.

Definition 13. *A* Multi-set pushdown system (Multi-set PDS) *is a WSPDS $((Q \times \mathbb{N}^k, \preceq), \Gamma, \delta)$, where*

- *Q, Γ are finite and $k = |\Gamma|$,*
- *δ is a finite set of transition rules consisting of two kinds:*
  1. *Increasing rules $\delta_1 : (p, \gamma, q, w, \boldsymbol{n})$ for $\boldsymbol{n} \in \mathbb{N}^k$;*
  2. *Decreasing rules δ_2: $(p, \bot, q, \bot, \boldsymbol{n})$ for $\boldsymbol{n} \in \mathbb{N}^k$.*

Configuration transitions are defined by:

$$\frac{(p, \gamma, q, w, \boldsymbol{n}) \in \delta_1}{\langle (p, \boldsymbol{m}), \gamma w'\rangle \hookrightarrow \langle (q, \boldsymbol{n} + \boldsymbol{m}), ww'\rangle} \qquad \frac{(p, \bot, q, \bot, \boldsymbol{n}) \in \delta_2, \boldsymbol{m} \geq \boldsymbol{n}}{\langle (p, \boldsymbol{m}), \bot\rangle \hookrightarrow \langle (q, \boldsymbol{m} - \boldsymbol{n}), \bot\rangle}$$

Note the decreasing rules are applied only when the stack is empty. A state in Pre_F^*-automata is in $\mathcal{I}(Q \times \mathbb{N}^k)$. Since Q is finite, we can always separate one state into finitely many states such that each of which has the form of $Q \times \mathcal{I}(\mathbb{N}^k)$. From Definition 12, we have two observations.

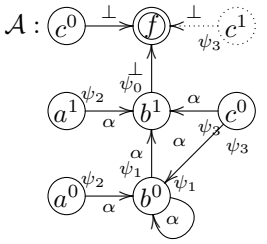
1. If transition $(p, K) \overset{\gamma}{\mapsto} s$ is added from $(q, K') \overset{w}{\Longmapsto} s$ by an increasing rule in δ_1, then $K \supseteq K'$.
2. If transition $(p, K) \overset{\bot}{\mapsto} s$ is added from $(q, K') \overset{\bot}{\mapsto} s$ by a decreasing rule in δ_2, then $K \subseteq K'$ and s is a final state.

Pre^*_F-saturation steps by increasing rules always enlarge ideals of vectors. By Lemma 1, eventually such ideals become maximal. Since stack alphabet is (finite thus) well-quasi-ordered, newly generated transitions by increasing rules are eventually caught by the first case of the $\oplus$ operator (in Definition 12). A worrying case is by decreasing rules, which shrink ideals. Since WQO does not guarantee the stabilization for $I_0 \supset I_1 \supset \cdots$, it may continue infinitely. For instance, Pre^*_F-saturation steps by decreasing pop rules may expand a path $\mapsto^*$ endlessly. Fortunately, decreasing rules of a Multi-set PDS occur only when the stack is empty. In such cases, destination states of $\mapsto$ are always final states, which are finitely many. Therefore, again they are eventually caught by the first case of the $\oplus$ operator. Note that this argument works even if we relax finite stack alphabet in Definition 13 to being well-quasi-ordered.

Corollary 2. *The* coverability problem *for a Multi-set PDS (with well-quasi-ordered stack alphabet) is decidable.*

Example 2. Let $\langle(\{a,b,c\} \times \mathbb{N}, \preceq), \{\alpha\}, \delta\rangle$ be a Multi-set PDS with transition rules given below. The set of configurations covering $\langle c^0, \bot\rangle$ is computed by Pre^*_F-automaton $\mathcal{A}$. We abbreviate ideal $\{p^n\}^\uparrow$ by p^n for $p \in \{a,b,c\}$ and $n \geq 0$. A transition $c^1 \overset{\bot}{\mapsto} f$ is generated from $a^1 \overset{\alpha\bot}{\Longmapsto} f$ by ψ_3. However, it is not added since we already have $c^0 \overset{\bot}{\mapsto} f$ and $\{c^1\}^\uparrow \subseteq \{c^0\}^\uparrow$.

$$\begin{array}{l}\delta_1 = \{\ \psi_1 : (b^{\boldsymbol{n}}, \alpha \to a^{\boldsymbol{n+1}}, \alpha),\\ \qquad\ \psi_2 : (a^{\boldsymbol{n}}, \alpha \to b^{\boldsymbol{n}}, \epsilon),\\ \qquad\ \psi_3 : (c^{\boldsymbol{n}}, \epsilon \to a^{\boldsymbol{n}}, \alpha)\}\\ \delta_2 = \{\ \psi_0 : (b^{\boldsymbol{n}}, \bot \to c^{\boldsymbol{n-1}}, \bot)\}\end{array}$$



5.3 Finite Control States

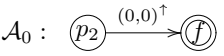
Assume that, for a monotonic WSPDS $M = \langle P, (\Gamma, \leq), \Delta\rangle$, P is finite and Δ does not contain nonstandard-pop rules. Then, we observe that, in the Pre^*_F-saturation for M, i) the set of states is bounded by the state in $\mathcal{A}_0$ and P, and ii) transitions between any pair of states are finitely many by Lemma 1. Hence, Pre^*_F saturation procedure finitely converges.

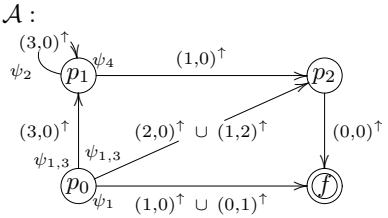
Theorem 4. *Let $\langle P, (\Gamma, \leq), \Delta\rangle$ be a WSPDS such that P is finite and $\psi^{-1}(I)$ is computable for any $(p, p', \psi) \in \Delta$. Then, its coverability is decidable.*

Example 3. Let $M = \langle \{p_i\}, \mathbb{N}^2, \Delta\rangle$ be a WSPDS with $\Delta = \{\psi_1, \psi_2, \psi_3, \psi_4\}$ given in the figure. An automaton $\mathcal{A}$ illustrates the pre^*-saturation starting from initial $\mathcal{A}_0$ that accepts $C = \langle p_2, (0,0)^\uparrow\rangle$.

For instance, $p_1 \overset{(3,0)^\uparrow}{\longmapsto} p_1$ in $\mathcal{A}$ is generated by ψ_2, and $p_0 \overset{(3,2)^\uparrow}{\longmapsto} p_1$ is added by ψ_3. Then repeatedly apply ψ_1 twice to $p_0 \overset{(3,2)^\uparrow}{\longmapsto} p_1 \overset{(3,0)^\uparrow}{\longmapsto} p_1$, we obtain $p_0 \overset{(3,0)^\uparrow}{\mapsto} p_1$.

$\psi_1 : \langle p_0, \boldsymbol{n}\rangle \to \langle p_0, (\boldsymbol{n} + (1,1))\boldsymbol{n}\rangle$
$\psi_2 : \langle p_1, \boldsymbol{n}\rangle \to \langle p_1, \epsilon\rangle$ if $\boldsymbol{n} \geq (3,0)$
$\psi_3 : \langle p_0, \boldsymbol{n}\rangle \to \langle p_1, \boldsymbol{n} - (0,2)\rangle$ if $\boldsymbol{n} \geq (0,2)$
$\psi_4 : \langle p_1, \boldsymbol{n}\rangle \to \langle p_2, \epsilon\rangle$ if $\boldsymbol{n} \geq (1,0)$





6 Conclusion

This paper investigated *well-structured pushdown systems* (WSPDSs), pushdown systems with well-quasi-ordered control states and stack alphabet, and developed two proof techniques to investigate the coverability based on extensions of classical P-automata techniques. They are,

- when a WSPDS has no standard push rules, the forward P-automata construction $Post^*$ with Karp-Miller acceleration, and
- when a WSPDS has no non-standard pop rules, the backward P-automata construction Pre^* with ideal representations.

We showed decidability results of coverability under certain conditions, which include *recursive vector addition system with states* [3], *multi-set pushdown systems* [20,13], and a WSPDS with finite control states and WQO stack alphabet. The first one extended the decidability of the state reachability in [3] to that of the coverability, and the second one relaxed finite stack alphabet of Multi-set PDSs [20,13] to being well-quasi-ordered.

Our current results just opened the possibility of WSPDSs. Among lots of things to do, we list few for future works.

- Currently, we have few examples of WSPDSs. For instance, parameterized systems would be good candidates to explore.
- Currently, we are mostly investigating with finite control states. However, we also found that a naive extension to infinite control states weakens the results a lot. We are looking for alternative conditions.
- Our decidability proofs contain algorithms to compute, however the estimation of their complexity is not easy due to the nature of WQO. We hope that a general theoretical observation [22] would give some hints.
- Our current forward method is restricted to VASs. We also hope to apply Finkel and Goubault-Larrecq's work on ω^2-WSTS [11,12] to generalize.

Acknowledgements. The authors would like to thank Prof.Alain Finkel and anonymous referees for valuable comments. This work is supported by the NSFC-JSPS bilateral joint research project (61011140074), NSFC projects (61003013,61100052,61033002), NSFC-ANR joint project (61261130589), and JSPS KAKENHI Grant-in-Aid for Scientific Research(B) (23300008).

References

1. Abdulla, P., Cerans, K., Jonsson, C., Yih-Kuen, T.: Algorithmic analysis of programs with well quasi-ordered domains. Information and Computation 160(1-2), 109–127 (2000)
2. Atig, M.F., Bouajjani, A., Qadeer, S.: Context-bounded analysis for concurrent pro-grams with dynamic creation of threads. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 107–123. Springer, Heidelberg (2009)
3. Bouajjani, A., Emmi, M.: Analysis of recursively parallel programs. In: Principles of Programming Languages (POPL 2012), pp. 203–214. ACM (2012)
4. 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)
5. Chadha, R., Viswanathan, M.: Decidability results for well-structured transition systems with auxiliary storage. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 136–150. Springer, Heidelberg (2007)
6. Demri, S., Jurdziński, M., Lachish, O., Lazic, R.: The covering and boundedness problems for branching vector addition systems. Journal of Computer and System Sciences 79(1), 23–38 (2012)
7. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)
8. Finkel, A.: A generalization of the procedure of Karp and Miller to well structured transition systems. In: Ottmann, T. (ed.) ICALP 1987. LNCS, vol. 267, pp. 499–508. Springer, Heidelberg (1987)
9. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. Electronic Notes Theoretical Computer Science 9, 27–37 (1997)
10. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science 256(1-2), 63–92 (2001)
11. Finkel, A., Goubault-Larrecq, J.: Forward analysis for WSTS, Part I: Completions. In: STACS 2009, pp. 433–444 (2009), `http://www.stacs-conf.org`
12. Finkel, A., Goubault-Larrecq, J.: Forward analysis for WSTS, Part II: Complete WSTS. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part II. LNCS, vol. 5556, pp. 188–199. Springer, Heidelberg (2009)
13. Jhala, R., Majumdar, R.: Interprocedural analysis of asynchronous programs. In: Principles of Programming Languages (POPL 2007), pp. 339–350. ACM (2007)
14. Lazić, R.: The reachability problem for vector addition systems with a stack is not elementary (2012), `http://rp12.labri.fr` (manuscript)
15. Leroux, J.: Vector addition system reachability problem. In: Principles of Programming Languages (POPL 2011), pp. 307–316. ACM (2011)
16. Mayr, E.: An algorithm for the general Petri net reachability problem. SIAM Journal Computing 13(3), 441–460 (1984)
17. Mayr, R.: Process rewrite systems. Information and Computation 156, 264–286 (1999)
18. 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)

19. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Programming Languages and Systems 22(2), 416–430 (2000)
20. 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)
21. Verma, K., Goubault-Larrecq, J.: Karp-Miller trees for a branching extension of VASS. Discrete Mathematics & Theoretical Computer Science 7(1), 217–230 (2005)
22. Weiermann, A.: Complexity bounds for some finite form of Kruskal's theorem. Journal of Symbolic Computation 18, 463–488 (1994)
23. Xiaojuan, C., Ogawa, M.: Well-structured pushdown system, Part 1: Decidable classes for coverability. JAIST Research Report IS-RR-2013-001 (2013), http://hdl.handle.net/10119/11347

A Relational Trace Logic for Vector Addition Systems with Application to Context-Freeness*

Jérôme Leroux, M. Praveen, and Grégoire Sutre

Univ. Bordeaux & CNRS, LaBRI, UMR 5800, Talence, France

Abstract. We introduce a logic for specifying trace properties of vector addition systems (VAS). This logic can express linear relations among pumping segments occurring in a trace. Given a VAS and a formula in the logic, we investigate the question whether the VAS contains a trace satisfying the formula. Our main contribution is an exponential space upper bound for this problem. The proof is based on a small model property for the logic. Compared to similar logics that are solvable in exponential space, a distinguishing feature of our logic is its ability to express non-context-freeness of the trace language of a VAS. This allows us to show that the context-freeness problem for VAS, whose complexity was not established so far, is ExpSpace-complete.

1 Introduction

Vector addition systems (VAS), or equivalently Petri nets, are well-studied for the modeling and analysis of concurrent systems. Despite their fairly large expressive power, many verification problems for VAS are decidable: coverability, boundedness, reachability, liveness, regularity, etc. [4]. The complexity of many of these decision problems has also been established.

Known decidable problems for VAS can be broadly classified into two classes. The first one, that we call $\mathcal{C}_{\mathrm{RP}}$, consists of those problems that are equivalent to the reachability problem in terms of computational complexity. Examples of problems in $\mathcal{C}_{\mathrm{RP}}$ include reachability, liveness, model checking some fragments of linear temporal logic, etc. The exact complexity of these problems is still open. The best known lower bound is ExpSpace-hardness [9]. There is no known upper bound, except that these problems are decidable [10,6].

The second class of problems, that we call $\mathcal{C}_{\mathrm{KM}}$, are those that can be decided using the Karp & Miller coverability graph [5]. Examples of problems in $\mathcal{C}_{\mathrm{KM}}$ include coverability, boundedness, regularity, etc. In general, the size of the coverability graph can be Ackermann in the size of the VAS. Still, most of the problems in $\mathcal{C}_{\mathrm{KM}}$ can be decided in exponential space, by applying a technique introduced by Rackoff [13], or extending this technique [1,2,3].

The question whether the set of traces of a VAS is context-free is also in the class $\mathcal{C}_{\mathrm{KM}}$. It was shown to be decidable by Schwer in [14], based on the coverability graph. Recently, we showed in [8] that non-context-freeness of the set of

* This work was supported by the ANR project ReacHard (ANR-11-BS02-001).

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 137–151, 2013.

traces can always be witnessed by a regular bounded language $u_1\sigma_1^* \cdots u_k\sigma_k^*$ that has a non-context-free intersection with the set of traces. Like context-freeness, for most of the properties in $\mathcal{C}_{\mathrm{KM}}$, we can find violating witnesses that are given by regular bounded languages of the form $u_1\sigma_1^* \cdots u_k\sigma_k^*$. Equivalently, we consider, in this paper, witnesses that are given as traces satisfying an iterability condition. Intuitively, a *self-covering sequence* is a trace $u_1\sigma_1 \cdots u_k\sigma_k$ such that, for every $n \geq 0$, there is a trace in $u_1\sigma_1^{\geq n} \cdots u_k\sigma_k^{\geq n}$. The words $\sigma_1, \ldots, \sigma_k$ are called *pumping segments*, as they can be iterated. To witness the violation of a given property, the displacements of the pumping segments are required to additionally satisfy some linear relations depending on the property under consideration.

Contributions. We introduce a relational logic over self-covering sequences. This logic can express positive Boolean combinations of linear relations among the displacements of pumping segments. We show that many properties in $\mathcal{C}_{\mathrm{KM}}$ can be expressed by the logic, in particular: coverability, boundedness, simultaneous unboundedness, regularity, and recurrence. Our main technical result is a small model property: we show that if there is a self-covering sequence satisfying a formula of the logic, then there is one of size at most doubly-exponential. This gives an exponential space upper bound for the problem whether a given VAS satisfies a given formula. Then, we focus on the context-freeness problem. We prove that the presence of self-covering sequences witnessing non-context-freeness can be expressed in our logic. We thus derive an exponential space upper bound for the context-freeness problem, whose complexity was still open.

Extensions of the technique introduced by Rackoff [13] are not enough for proving the small model property mentioned above. Our approach is based on reversibility domains [7]. The reversibility domain of an action is the set of configurations from which the effect of the action can be canceled by a word of actions. A doubly exponential bound on the minimal elements of these upward closed sets is derived in [7]. This result is central in our approach.

Related Work. Other logics that can be checked in exponential space have been investigated before. The fragment of Yen's path logic [17] introduced by Atig and Habermehl [1], the fragment of computational tree logic by Blockelet and Schmitz [2] and the generalized unboundedness properties of Demri [3] are in this category. All of these impose conditions that are incompatible with context-freeness. We provide a more detailed comparison with related work at the end of the paper.

Outline. We recall in Section 2 some basic notions on VAS and define self-covering sequences. Section 3 introduces our relational trace logic. We show in Section 4 that many classical problems on VAS can be expressed in this logic. We establish in Sections 5, 6, and 7 the exponential space complexity of the problem whether a given VAS satisfies a given formula. Section 8 applies the results of the previous sections to the context-freeness problem for VAS. We conclude in Section 9 with a detailed comparison with related work.

2 Vector Addition Systems

We let $\mathbb{N}$ and $\mathbb{Z}$ denote the sets of *natural numbers* and *integers* respectively. For $\mathbb{X} \in \{\mathbb{N}, \mathbb{Z}\}$ and $\# \in \{<, \leq, \geq, >\}$, we write $\mathbb{X}_{\#0} = \{x \in \mathbb{X} \mid x \# 0\}$. Vectors and sets of vectors are typeset in bold face. The ith *component* of a vector $\boldsymbol{v}$ is written $\boldsymbol{v}(i)$. The *zero vector* is written $\mathbf{0}$. We let $\boldsymbol{e}_i$ denote the ith *unit vector*, defined by $\boldsymbol{e}_i(i) = 1$ and $\boldsymbol{e}_i(j) = 0$ for every index $j \neq i$. Given a vector $\boldsymbol{v}$, we write $\|\boldsymbol{v}\|^+, \|\boldsymbol{v}\|^-$ and $\|\boldsymbol{v}\|^0$ for the sets of indices i such that $\boldsymbol{v}(i) > 0, \boldsymbol{v}(i) < 0$ and $\boldsymbol{v}(i) = 0$, respectively. We denote by $\|\boldsymbol{v}\|_\infty$ the *infinite norm* $\max_i |\boldsymbol{v}(i)|$. Given a finite set $\boldsymbol{V}$ of vectors, we introduce $\|\boldsymbol{V}\|_\infty = \max_{\boldsymbol{v} \in \boldsymbol{V}} \|\boldsymbol{v}\|_\infty$. A *word* is a finite sequence $\sigma = \boldsymbol{v}_1 \cdots \boldsymbol{v}_n$ of vectors in $\mathbb{Z}^d$. We let $|\sigma|$ denote the *length* n of the word σ. The *displacement* of σ is the sum $\sum_{j=1}^{n} \boldsymbol{v}_j$, denoted by $\Delta(\sigma)$.

We now recall the main concepts of vector addition systems (VAS). Consider a *dimension* $d \in \mathbb{N}$, with $d > 0$. A *configuration* is a vector $\boldsymbol{c} \in \mathbb{N}^d$, and an *action* is a vector $\boldsymbol{a} \in \mathbb{Z}^d$. Informally, a vector addition system moves from one configuration to the next by adding an action. This operational semantics is formalized by the labeled transition relation $\rightarrow \subseteq \mathbb{N}^d \times \mathbb{Z}^d \times \mathbb{N}^d$ defined by $\boldsymbol{c} \xrightarrow{\boldsymbol{a}} \boldsymbol{c'}$ if $\boldsymbol{c'} = \boldsymbol{c} + \boldsymbol{a}$. In particular, notice that an action $\boldsymbol{a}$ is enabled in a configuration $\boldsymbol{c}$ if, and only if, $\boldsymbol{c} + \boldsymbol{a} \geq \mathbf{0}$. A *run* is a finite, alternating sequence $(\boldsymbol{c}_0, \boldsymbol{a}_1, \boldsymbol{c}_1, \ldots, \boldsymbol{a}_n, \boldsymbol{c}_n)$ of configurations and actions, satisfying $\boldsymbol{c}_{i-1} \xrightarrow{\boldsymbol{a}_i} \boldsymbol{c}_i$ for all i. We write $\boldsymbol{c}_0 \xrightarrow{\boldsymbol{a}_1 \cdots \boldsymbol{a}_n} \boldsymbol{c}_n$ when the intermediate configurations are not important. The word $\boldsymbol{a}_1 \cdots \boldsymbol{a}_n$ is called the *label* of the run. A *trace* from a configuration $\boldsymbol{c}$ is the label of some run that starts with $\boldsymbol{c}$. Given an *initial configuration* $\boldsymbol{c}_{\text{init}} \in \mathbb{N}^d$, we let $\mathcal{T}(\boldsymbol{c}_{\text{init}})$ denote the set of all traces from $\boldsymbol{c}_{\text{init}}$.

A *vector addition system* is a pair $\langle \boldsymbol{A}, \boldsymbol{c}_{\text{init}} \rangle$ where $\boldsymbol{A}$ is a finite subset of $\mathbb{Z}^d$ and $\boldsymbol{c}_{\text{init}} \in \mathbb{N}^d$ is an initial configuration. Its operational semantics is obtained by restricting the labeled transition relation $\rightarrow$ to actions in $\boldsymbol{A}$. Accordingly, a *trace* of a VAS $\langle \boldsymbol{A}, \boldsymbol{c}_{\text{init}} \rangle$ is a trace from $\boldsymbol{c}_{\text{init}}$ that is contained in $\boldsymbol{A}^*$. The set of all traces of $\langle \boldsymbol{A}, \boldsymbol{c}_{\text{init}} \rangle$, written $\mathcal{T}(\boldsymbol{A}, \boldsymbol{c}_{\text{init}}) = \mathcal{T}(\boldsymbol{c}_{\text{init}}) \cap \boldsymbol{A}^*$, is called the *trace language* of $\langle \boldsymbol{A}, \boldsymbol{c}_{\text{init}} \rangle$.

In this paper, we consider verification properties that can be checked through witnesses that are traces satisfying some pumping conditions. These are called self-covering sequences, and defined as follows.

Definition 2.1. *A* self-covering sequence *for a VAS* $\langle \boldsymbol{A}, \boldsymbol{c}_{init} \rangle$ *is a non-empty sequence* $(u_1, \sigma_1, \ldots, u_k, \sigma_k)$ *of words such that* $u_1\sigma_1 \cdots u_k\sigma_k$ *is a trace and* $\|\Delta(\sigma_h)\|^- \subseteq \bigcup_{j=1}^{h-1} \|\Delta(\sigma_j)\|^+$ *for all* $h \in \{1, \ldots, k\}$.

The words $\sigma_1, \ldots, \sigma_k$ are called *pumping segments*, k is the *number of pumping segments*, and $|u_1|+|\sigma_1|+\cdots+|u_k|+|\sigma_k|$ is the *size* of the self-covering sequence. If $k = 1$ in the above definition and $\|\Delta(\sigma_1)\|^+ \neq \emptyset$, we get the standard self-covering sequences, which are known to witness unboundedness of VAS [5]. The next lemma states a property of self-covering sequences that explains the name given to the words $\sigma_1, \ldots, \sigma_k$.

Lemma 2.2. *A sequence $(u_1, \sigma_1, \ldots, u_k, \sigma_k)$ of words is self-covering for a VAS $\langle \boldsymbol{A}, \boldsymbol{c}_{init} \rangle$ if, and only if, $u_1\sigma_1 \ldots u_k\sigma_k$ is a trace and for every $n \in \mathbb{N}$, there exist $n_1, \ldots, n_k \geq n$ such that $u_1\sigma_1^{n_1} \cdots u_k\sigma_k^{n_k}$ is a trace.*

3 A Relational Logic to Express Properties of Traces

In this section, we introduce a logic that can express properties of VAS such as unboundedness, place unboundedness, and non-regularity (see Section 4 for examples). The logic has terms t and formulas ϕ of the following syntax:

$$t ::= z\boldsymbol{\delta}_j(i) \mid t + t, \quad z \in \mathbb{Z}, j \geq 1, 1 \leq i \leq d$$
$$\phi ::= t \geq n \mid \phi \vee \phi \mid \phi \wedge \phi, \quad n \in \mathbb{N}$$

In the above syntax, $\boldsymbol{\delta}_j$ are variables that have to be interpreted. The norm $\|t\|_1$ of a term t is defined inductively as follows: $\|z\boldsymbol{\delta}_j(i)\|_1 = |z|$, $\|t_1 + t_2\|_1 = \|t_1\|_1 + \|t_2\|_1$. The norm $\|\phi\|_1$ of a formula ϕ is defined by $\|t \geq n\|_1 = \|t\|_1 + n$, $\|\phi_1 \vee \phi_2\|_1 = \|\phi_1 \wedge \phi_2\|_1 = \|\phi_1\|_1 + \|\phi_2\|_1$.

Definition 3.1. *A self-covering sequence $(u_1, \sigma_1, \ldots, u_k, \sigma_k)$ satisfies a formula ϕ if ϕ is true according to the usual laws of arithmetic when $\boldsymbol{\delta}_j$ is set to $\Delta(\sigma_j)$ for $j \leq k$ and $\boldsymbol{\delta}_j$ is set to $\mathbf{0}$ for $j > k$.*

A VAS satisfies a formula ϕ if it admits a self-covering sequence satisfying ϕ. The *model-checking* problem for this logic asks whether a given VAS satisfies a given formula.

Remark 3.2. The satisfaction of the formula does not depend on the words $u_1, \ldots, u_k$. However, without these words, the reachability problem for vector addition systems can be easily reduced to the model-checking problem for the logic. Recall that the reachability problem consists in deciding if a given configuration is the last configuration of a run starting from the initial one. This problem is known to be decidable[10,6] but no complexity upper-bound is known. An adaptation of the proof of [1, Theorem 3] shows that the reachability problem for VAS can be reduced to the model-checking problem for our logic by additionally requiring that $u_1, \ldots, u_k$ are empty words. □

The model-checking problem for our logic can be solved by constructing the Karp & Miller coverability graph [5]. However, the size of the coverability graph can be Ackermann in the size of the VAS. We will show in Sections 5 up to 7 that this problem can be solved in exponential space. Before that, let us present some applications of our logic.

4 Examples and Short Extensions

In this section, we show that classical problems can be reduced to the model-checking problem for our logic. We prove that unboundedness, place unboundedness and non-regularity can be directly encoded with formulas. We also provide

short extensions of the logic based on simple encodings that can express recurrence and coverability. All these problems are recalled in this section.

We first present problems that can be directly reduced to the model-checking problem for our logic. Recall that a configuration $\boldsymbol{c}$ is *reachable* if there is a run from the initial configuration to $\boldsymbol{c}$. The set of reachable configurations is called the *reachability set*. A vector addition system is *bounded* if its reachability set is finite. The boundedness problem was proved to be decidable by Karp and Miller in [5]. The decidability comes from a characterization of unbounded vector addition systems; a vector addition system is unbounded if, and only if, there exists a self-covering sequence (u_1, σ_1) such that $\|\Delta(\sigma_1)\|^+ \neq \emptyset$. Karp and Miller provided a way for deciding this property based on the computation of a tree (the Karp & Miller coverability tree). The complexity of this algorithm is non-primitive recursive [11]. Lipton proved in [9] that the boundedness problem requires exponential space. In [13], Rackoff provided an exponential space upper bound based on a doubly-exponential bound on self-covering sequences witnessing unboundedness. We observe that a vector addition system is unbounded if, and only if, it satisfies the following formula:

$$\bigvee_{i=1}^{d} \boldsymbol{\delta}_1(i) \geq 1$$

The boundedness problem was generalized by introducing variants like the place boundedness problem that asks which components (also called places for Petri nets) are unbounded. The place boundedness problem requires exponential space. The proof is by a simple reduction from the boundedness problem. Whereas the place boundedness problem was considered in different papers, no upper bound of complexity was published until recently in [3]. In that paper, Demri introduced a more general problem, useful for reducing different problems, called the simultaneous unboundedness problem. A vector addition system is *simultaneously unbounded* on a set $I \subseteq \{1, \ldots, d\}$ of indexes, if, for every bound $b \in \mathbb{N}$, there exists a reachable configuration $\boldsymbol{c}$ such that $\boldsymbol{c}(i) \geq b$ for every $i \in I$. Demri proved that a vector addition system is simultaneously unbounded on I if, and only if, it satisfies the following formula:

$$\bigwedge_{i \in I} \bigvee_{j=1}^{d} \boldsymbol{\delta}_j(i) \geq 1$$

A vector addition system is called *regular* when its trace language is regular. In [16], Valk and Vidal-Naquet provided a characterization of non-regularity for vector addition systems. Since the characterization is based on the Karp & Miller coverability graph [5], the Valk and Vidal-Naquet algorithm is non-primitive recursive. In [1], Atig and Habermehl observed that the regularity problem cannot be expressed in their fragment of Yen's path logic that is decidable in exponential space, and left the complexity open. Based on the simultaneous unboundedness approach, Demri proved in [3] that the regularity problem is decidable in

exponential space. This upper bound is obtained by observing that the trace language of a VAS is non-regular if, and only if, the VAS satisfies the following formula:

$$\bigvee_{i=1}^{d} -\boldsymbol{\delta}_{d+1}(i) \geq 1$$

Till now, we proved that some classical problems can be reduced to the model-checking problem for our logic. For other problems, we need short extensions that require simple encodings. In the remainder of this section, we show the kind of extensions that can be useful for deciding recurrence and coverability problems.

A set of actions $\boldsymbol{T}$ of a vector addition system is said to be *recurrent* if there exists a self-covering sequence (u_1, σ_1) such that $\boldsymbol{T}$ is the set of actions occurring in σ_1. The verification of LTL properties and some other properties like promptness detection (see, e.g., [1,15]) can be reduced to the recurrence problem. The latter problem can be reduced to the model-checking problem for our logic by introducing extra components, one for each action, counting the number of times an action is executed. Let us consider a VAS $\langle \boldsymbol{A}, \boldsymbol{c}_{\text{init}} \rangle$ and a subset $\boldsymbol{T} \subseteq \boldsymbol{A}$. We assume that $\boldsymbol{A} = \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\}$. Recall that $\boldsymbol{e}_\ell$ is the unit vector defined by $\boldsymbol{e}_\ell(\ell) = 1$ and $\boldsymbol{e}_\ell(i) = 0$ if $i \neq \ell$. We introduce the VAS $\langle \boldsymbol{A}', \boldsymbol{c}'_{\text{init}} \rangle$ of dimension $d + n$ defined by $\boldsymbol{A}' = \{(\boldsymbol{a}_\ell, \boldsymbol{e}_\ell) \mid 1 \leq \ell \leq n\}$ and $\boldsymbol{c}'_{\text{init}} = (\boldsymbol{c}_{\text{init}}, \boldsymbol{0})$. Observe that $\boldsymbol{T}$ is recurrent for the VAS $\langle \boldsymbol{A}, \boldsymbol{c}_{\text{init}} \rangle$ if, and only if, $\langle \boldsymbol{A}', \boldsymbol{c}'_{\text{init}} \rangle$ satisfies the following formula:

$$\bigwedge_{j \mid \boldsymbol{a}_j \in \boldsymbol{T}} \boldsymbol{\delta}_1(d+j) \geq 1 \quad \wedge \quad \bigwedge_{j \mid \boldsymbol{a}_j \notin \boldsymbol{T}} -\boldsymbol{\delta}_1(d+j) \geq 0$$

The same transformation provides a simple way for encoding more complex relations between numbers of occurrences in different pumping segments of self-covering sequences. For instance, the strong promptness detection (see, e.g., [1]) can be encoded with the previous formula by replacing $\boldsymbol{\delta}_1$ by $\boldsymbol{\delta}_d$.

One can also check coverability properties with the help of an additional component. Recall that a configuration $\boldsymbol{c} \in \mathbb{N}^d$ is *coverable* if there exists a reachable configuration larger than or equal to $\boldsymbol{c}$, i.e., a reachable configuration in $\boldsymbol{c} + \mathbb{N}^d$. The coverability problem asks whether a given configuration is coverable in a given VAS. Lipton derived an exponential space lower bound in [9] and Rackoff provided an exponential space upper bound in [13]. The coverability problem can be reduced to the place boundedness problem as follows. Given a vector addition system $\langle \boldsymbol{A}, \boldsymbol{c}_{\text{init}} \rangle$ and a configuration $\boldsymbol{c}$, we consider the vector addition system $\langle \boldsymbol{A}', \boldsymbol{c}'_{\text{init}} \rangle$ defined by $\boldsymbol{A}' = (\boldsymbol{A} \times \{0\}) \cup \{(-\boldsymbol{c}, 2), (\boldsymbol{c}, -1)\}$ and $\boldsymbol{c}'_{\text{init}} = (\boldsymbol{c}_{\text{init}}, 0)$. Just observe that $\boldsymbol{c}$ is coverable in $\langle \boldsymbol{A}, \boldsymbol{c}_{\text{init}} \rangle$ if, and only if, the last component of the VAS $\langle \boldsymbol{A}', \boldsymbol{c}'_{\text{init}} \rangle$ is unbounded. Therefore, the coverability problem can be reduced to the model-checking problem for our logic. With a similar transformation, we can encode more complex properties that require multiple coverings along the pumping segments of a self-covering sequence.

5 Small Model Property

In this section, we show that if there is a self-covering sequence satisfying a formula, there is one whose length is bounded in terms of the sizes of the VAS and the formula. As a consequence, we get ExpSpace-completeness for the model-checking problem for our logic.

The bound we give for the size of satisfying self-covering sequences depends in a specific way on how conjunctions are distributed in a formula. We define below two measures of formulas that will be used in our bound.

Definition 5.1. *For a formula ϕ, the* conjunction rank $r(\phi)$ *is defined inductively as follows: $r(t \geq n) = 1$, $r(\phi_1 \vee \phi_2) = \max\{r(\phi_1), r(\phi_2)\}$ and $r(\phi_1 \wedge \phi_2) = r(\phi_1) + r(\phi_2)$. By $k(\phi)$ we denote the maximal j such that $\boldsymbol{\delta}_j$ occurs in ϕ.*

Intuitively, $r(\phi)$ is a bound on the number of terms that need to be satisfied simultaneously to satisfy ϕ.

Theorem 5.2. *If there is a self-covering sequence in $\langle \boldsymbol{A}, \boldsymbol{c}_{init} \rangle$ satisfying ϕ, there is one of size at most $(\|\boldsymbol{A}\|_\infty + \|\phi\|_1)^{r(\phi)c^{(d \cdot k(\phi))^3}}$ where c is a constant.*

The proof is in two parts. The first part is bounding the lengths of $u_1, \ldots, u_k$ that occur in between the pumping segments $\sigma_1, \ldots, \sigma_k$. We will make use of the following result, which is an easy consequence of some proofs given in [13].

Lemma 5.3. *Suppose that $\boldsymbol{c}_{init} \xrightarrow{\sigma} \boldsymbol{c}_1$ and $\boldsymbol{c}_1 \geq \boldsymbol{c}$. Then there is a sub-word σ' of σ such that $\boldsymbol{c}_{init} \xrightarrow{\sigma'} \boldsymbol{c}'_1$, $\boldsymbol{c}'_1 \geq \boldsymbol{c}$ and $|\sigma'| \leq (\|\boldsymbol{A}\|_\infty + \|\boldsymbol{c}\|_\infty)^{(d+1)!}$.*

Proof. Follows easily from a close observation of [13, Proof of Lemma 3.4]. □

Lemma 5.4. *Consider a run $\boldsymbol{c}_{init} \xrightarrow{u_1} \boldsymbol{c}_1 \xrightarrow{\sigma_1} \boldsymbol{c}'_1 \to \cdots \xrightarrow{u_k} \boldsymbol{c}_k \xrightarrow{\sigma_k} \boldsymbol{c}'_k$ of $\langle \boldsymbol{A}, \boldsymbol{c}_{init} \rangle$, with $|\sigma_1| + \cdots + |\sigma_k| \leq l$. Then there are words $u'_1, \ldots, u'_k$ such that $u'_1\sigma_1 \cdots u'_k\sigma_k$ is a trace and $|u'_1| + \cdots + |u'_k| \leq (2l\|\boldsymbol{A}\|_\infty)^{((d+1)k+1)!}$.*

Proof (Sketch). For any word σ, let $\boldsymbol{c}_\sigma$ be the unique minimal configuration that enables σ. Since $\boldsymbol{c}_j \geq \boldsymbol{c}_{\sigma_j}$ for all j, $1 \leq j \leq k$, $\boldsymbol{c}_{\sigma_j}$ are all coverable from $\boldsymbol{c}_{\text{init}}$. Let $\boldsymbol{c}^k_{\text{init}}$ be the vector obtained by adjoining k copies of $\boldsymbol{c}_{\text{init}}$ and let $\boldsymbol{c}''$ be the vector obtained by adjoining $\boldsymbol{c}_{\sigma_1}, \ldots, \boldsymbol{c}_{\sigma_k}$. We can now think of a suitably defined new VAS where $\boldsymbol{c}''$ is coverable from the initial configuration $\boldsymbol{c}^k_{\text{init}}$. From Lemma 5.3, we infer that there is a sub-word of the original covering sequence that also covers $\boldsymbol{c}''$, whose length is bounded. From this short covering sequence, we extract words $u'_1, \ldots, u'_k$ of the original VAS satisfying the length requirements. □

Now it is enough to bound the length of the pumping segments. Suppose $\boldsymbol{c}_{\text{init}} \xrightarrow{u_1} \boldsymbol{c}_1 \xrightarrow{\sigma_1} \boldsymbol{c}'_1 \to \cdots \xrightarrow{u_k} \boldsymbol{c}_k \xrightarrow{\sigma_k} \boldsymbol{c}'_k$. Indices in $\|\sigma_1\|^+$ can potentially reach arbitrarily high values (by repeating σ_1 many times). We want to momentarily forget the exact value of these indices and emphasize that they can be as large as needed. This is done by allowing values to be ω.

Definition 5.5. *Let* $\mathbb{N}_\omega = \mathbb{N} \cup \{\omega\}$. *Let* $\omega \geq n$ *and* $\omega - n = \omega + n = \omega$ *for all* $n \in \mathbb{N}$. *An* extended configuration *is a vector* $\boldsymbol{x} \in \mathbb{N}_\omega^d$. *The labeled transition relation* $\rightarrow$ *is extended to* $\rightarrow \subseteq \mathbb{N}_\omega^d \times \mathbb{Z}^d \times \mathbb{N}_\omega^d$ *defined by* $\boldsymbol{x} \xrightarrow{\boldsymbol{a}} \boldsymbol{x}'$ *if* $\boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{a}$. *We denote by* $\|\boldsymbol{x}\|^\omega$ *the set of indices* i *such that* $\boldsymbol{x}(i) = \omega$.

Let $\boldsymbol{x}_1$ be the extended configuration that is the same as $\boldsymbol{c}_1$ except in indices that are increased by σ_1, where $\boldsymbol{x}_1$ has ω. That is, let $\boldsymbol{x}_1(i) = \omega$ for $i \in \|\Delta(\sigma_1)\|^+$ and $\boldsymbol{x}_1(i) = \boldsymbol{c}_1'(i) = \boldsymbol{c}_1(i)$ for $i \in \|\Delta(\sigma_1)\|^0$ ($\|\Delta(\sigma_1)\|^- = \emptyset$). Similarly, let $\boldsymbol{x}_2(i) = \omega$ for $i \in \|\sigma_1\|^+ \cup \|\sigma_2\|^+$ and $\boldsymbol{x}_2(i) = \boldsymbol{c}_2'(i) = \boldsymbol{c}_2(i)$ for $i \in \|\Delta(\sigma_2)\|^0 \setminus \|\sigma_1\|^+$ ($\|\Delta(\sigma_2)\|^- \subseteq \|\Delta(\sigma_1)\|^+$). The extended configurations $\boldsymbol{x}_3, \ldots, \boldsymbol{x}_k$ are similar.

We have $\boldsymbol{x}_j \xrightarrow{\sigma_j} \boldsymbol{x}_j$ for all j, $1 \leq j \leq k$, which can be thought of as $\boldsymbol{x} \xrightarrow{\sigma} \boldsymbol{x}$ in a suitably defined (kd)-dimensional VAS. Hence, σ is a cycle on $\boldsymbol{x}$ in this new VAS. Note that $\Delta(\sigma)$ is not necessarily $\mathbf{0}$, since $\boldsymbol{x}$ may have some omega components. The fact that $\sigma_1, \ldots, \sigma_k$ are pumping segments satisfying ϕ can be encoded into a linear system of the form $\boldsymbol{Z}\Delta(\sigma) \geq \boldsymbol{n}$. We prove in the next section that if there are cycles satisfying such a condition, there will be similar cycles of bounded length. From such a short cycle, we can extract words $\sigma_1', \ldots, \sigma_k'$ of the original VAS meeting the length requirements of Theorem 5.2.

6 Short Cycles via Reversibility Domains

In this section, we show that for every cycle $\boldsymbol{x} \xrightarrow{\sigma} \boldsymbol{x}$ satisfying a linear system, there exists a similar short cycle $\boldsymbol{x} \xrightarrow{\sigma'} \boldsymbol{x}$. The proof is based on the following two theorems providing bounds related to reversible words. These results are proved in [7]. Given an implicit VAS $\langle \boldsymbol{A}, \boldsymbol{c}_{\text{init}} \rangle$, a word $u \in \boldsymbol{A}^*$ is said to be *reversible* on an extended configuration $\boldsymbol{c}$ if there exists a word $v \in \boldsymbol{A}^*$ such that $\boldsymbol{c} \xrightarrow{uv} \boldsymbol{c}$ and such that $\Delta(uv) = \mathbf{0}$. The *domain of reversibility* of an action $\boldsymbol{a} \in \boldsymbol{A}$ is the set of extended configurations $\boldsymbol{c}$ on which $\boldsymbol{a}$ is reversible.

Theorem 6.1 ([7, Theorem 10.1]). *Let* $u \in \boldsymbol{A}^*$ *be a reversible word on an extended configuration* $\boldsymbol{c}$. *There exists* $u' \in \boldsymbol{A}^*$ *reversible on* $\boldsymbol{c}$ *such that* $\Delta(u) = \Delta(u')$ *and* $|u'| \leq 17d^2 x^{15d^{d+2}}$, *where* $x = (1 + \|\boldsymbol{A}\|_\infty)(1 + \|\boldsymbol{c}\|_\infty + \|\Delta(u)\|_\infty)$.

Theorem 6.2 ([7, Theorem 11.1]). *For every extended configuration* $\boldsymbol{c}$ *in the domain of reversibility of an action* $\boldsymbol{a} \in \boldsymbol{A}$, *there exists a configuration* $\boldsymbol{c}' \leq \boldsymbol{c}$ *in the domain of reversibility of* $\boldsymbol{a}$ *such that* $\|\boldsymbol{c}'\|_\infty \leq (102d^2\|\boldsymbol{A}\|_\infty^2)^{(15d^{d+2})^{d+2}}$.

To show the existence of such short cycles, we need to introduce some notations and a technical result regarding minimal solutions of linear diophantine systems. For an integer vector $\boldsymbol{v}$, let $\|\boldsymbol{v}\|_1$ denote the sum $\sum_i |\boldsymbol{v}(i)|$. For a finite set of vectors $\boldsymbol{V}$, $\|\boldsymbol{V}\|_1$ denotes $\max_{\boldsymbol{v} \in \boldsymbol{V}} \{\|\boldsymbol{v}\|_1\}$. For an integer matrix $\boldsymbol{Z}$, let $\|\boldsymbol{Z}\|_{1,\infty}$ denote $\max_i \{\sum_j |\boldsymbol{Z}(i,j)|\}$. For $z \in \mathbb{Z}$, let $z\boldsymbol{v}$ denote the vector such that $(z\boldsymbol{v})(i) = z \cdot \boldsymbol{v}(i)$ for all i. Let $\mathbb{N}\boldsymbol{v}$ denote the set of vectors $\{n\boldsymbol{v} \mid n \in \mathbb{N}\}$. For two sets of vectors $\boldsymbol{V}_1, \boldsymbol{V}_2$ of the same dimension, let $\boldsymbol{V}_1 + \boldsymbol{V}_2$ denote the set of vectors $\{\boldsymbol{v}_1 + \boldsymbol{v}_2 \mid \boldsymbol{v}_1 \in \boldsymbol{V}_1, \boldsymbol{v}_2 \in \boldsymbol{V}_2\}$.

Based on [12], one can easily derive the following lemma.

Lemma 6.3. *Let $\boldsymbol{Z}$ be a $r \times d$ integer matrix and let $\boldsymbol{b} \in \mathbb{Z}^r$ be a vector. The set of all integer vectors $\boldsymbol{\rho}$ such that $\boldsymbol{Z}\boldsymbol{\rho} \geq \boldsymbol{b}$ is a finite union of sets of the form $\boldsymbol{p}_0 + \mathbb{N}\boldsymbol{p}_1 + \cdots + \mathbb{N}\boldsymbol{p}_m$, where $m \in \mathbb{N}$ and $\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots, \boldsymbol{p}_m$ are integer vectors such that $\|\boldsymbol{p}_0\|_1, \|\boldsymbol{p}_1\|_1, \ldots, \|\boldsymbol{p}_m\|_1 \leq (2 + \|\boldsymbol{Z}\|_{1,\infty} + \|\boldsymbol{b}\|_\infty)^r$.*

Now we are ready to prove the existence of short cycles.

Lemma 6.4. *Let $\boldsymbol{A} \subseteq \mathbb{Z}^d$ be a finite set and $\boldsymbol{x} \in \mathbb{N}_\omega^d$ be an extended configuration. Suppose there is a word $\sigma \in \boldsymbol{A}^*$ such that $\boldsymbol{x} \xrightarrow{\sigma} \boldsymbol{x}$ and $\boldsymbol{Z}\Delta(\sigma) \geq \boldsymbol{n}$, where $\boldsymbol{Z} \in \mathbb{Z}^{r \times d}$ is an integer matrix with r rows, d columns and $\boldsymbol{n} \in \mathbb{N}^r$ is a vector of natural numbers. Then there is a word $\sigma' \in \boldsymbol{A}^*$ such that $\boldsymbol{x} \xrightarrow{\sigma'} \boldsymbol{x}$, $\boldsymbol{Z}\Delta(\sigma') \geq \boldsymbol{n}$ and $|\sigma'| \leq (\|\boldsymbol{Z}\|_{1,\infty} + \|\boldsymbol{A}\|_\infty + \|\boldsymbol{n}\|_\infty)^{rc_2^{d^3}}$ for some constant c_2.*

Proof. Since $\boldsymbol{x} \xrightarrow{\sigma} \boldsymbol{x}$, $\Delta(\sigma)(i) = 0$ for all $i \notin \|\boldsymbol{x}\|^\omega$. We can encode these conditions as additional inequalities in $\boldsymbol{Z}\Delta(\sigma) \geq \boldsymbol{n}$, by adding at most $2d$ rows to $\boldsymbol{Z}$ and $\boldsymbol{n}$. Let $\boldsymbol{Z}'\Delta(\sigma) \geq \boldsymbol{n}'$ be the resulting system. By Lemma 6.3, the set of all vectors $\boldsymbol{\rho} \in \mathbb{Z}^d$ satisfying $\boldsymbol{Z}'\boldsymbol{\rho} \geq \boldsymbol{n}'$ is a finite union of sets of the form $\boldsymbol{p}_0 + \mathbb{N}\boldsymbol{p}_1 + \cdots + \mathbb{N}\boldsymbol{p}_m$, where $m \in \mathbb{N}$ and $\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots, \boldsymbol{p}_m$ are integer vectors such that $\|\boldsymbol{p}_0\|_1, \|\boldsymbol{p}_1\|_1, \ldots, \|\boldsymbol{p}_m\|_1 \leq (2 + \|\boldsymbol{Z}\|_{1,\infty} + \|\boldsymbol{n}'\|_\infty)^{r+2d}$. Since $\boldsymbol{Z}'\boldsymbol{p}_0 \geq \boldsymbol{n}'$ and $\boldsymbol{n}'$ is a vector of natural numbers, we have $\boldsymbol{Z}'(i\boldsymbol{p}_0) \geq \boldsymbol{n}'$ for all $i \geq 1$. Hence, we can assume without loss of generality that the sets are of the form $\boldsymbol{p}_0 + \mathbb{N}\boldsymbol{p}_0 + \mathbb{N}\boldsymbol{p}_1 + \cdots + \mathbb{N}\boldsymbol{p}_m$. Since $\boldsymbol{Z}'\boldsymbol{p}_0 \geq \boldsymbol{n}'$, $\boldsymbol{p}_0(i) = 0$ for $i \notin \|\boldsymbol{x}\|^\omega$. Since $\boldsymbol{Z}'(\boldsymbol{p}_0 + \boldsymbol{p}_j) \geq \boldsymbol{n}'$ for all $j \in \{1, \ldots, m\}$, $(\boldsymbol{p}_0 + \boldsymbol{p}_j)(i) = 0$ for $i \notin \|\boldsymbol{x}\|^\omega$. Hence, $\boldsymbol{p}_j(i) = 0$ for $i \notin \|\boldsymbol{x}\|^\omega$ and $j \in \{1, \ldots, m\}$. In words, this means that vectors $\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots, \boldsymbol{p}_m$ have value 0 in indices that are not ω in $\boldsymbol{x}$.

Suppose $\Delta(\sigma) = \boldsymbol{p}_0 + i_0\boldsymbol{p}_0 + \cdots + i_m\boldsymbol{p}_m$. Let $\widetilde{\boldsymbol{A}} = \boldsymbol{A} \cup \{-\boldsymbol{p}_0, \ldots, -\boldsymbol{p}_m\}$. We have $\boldsymbol{x} \xrightarrow{-\boldsymbol{p}_0} \boldsymbol{x} \xrightarrow{(-\boldsymbol{p}_0)^{i_0} \cdots (-\boldsymbol{p}_m)^{i_m}\sigma} \boldsymbol{x}$, written as $\boldsymbol{x} \xrightarrow{-\boldsymbol{p}_0} \boldsymbol{x} \xrightarrow{u} \boldsymbol{x}$ for simplicity. By our choice of $i_0, \ldots, i_m$, we get $\Delta(u) = \boldsymbol{p}_0$. This means the action $-\boldsymbol{p}_0$ is *reversible* on the extended configuration $\boldsymbol{x}$. Hence, by Theorem 6.2, there exists a configuration $\boldsymbol{c}' \leq \boldsymbol{x}$ such that $-\boldsymbol{p}_0$ is reversible on $\boldsymbol{c}'$ and $\|\boldsymbol{c}'\|_\infty \leq (102d^2\tilde{a}^2)^{(15d^{d+2})^{d+2}}$, where $\tilde{a} = \|\widetilde{\boldsymbol{A}}\|_\infty$. Now we have $\boldsymbol{c}' \xrightarrow{-\boldsymbol{p}_0} \boldsymbol{c}'' \xrightarrow{u'} \boldsymbol{c}'$, so $\Delta(u') = \boldsymbol{p}_0$. The word u' is reversible on the configuration $\boldsymbol{c}''$. By Theorem 6.1, there exists another word u'' reversible on $\boldsymbol{c}''$ such that $\Delta(u'') = \Delta(u') = \boldsymbol{p}_0$ and $|u''| \leq 17d^2x^{15d^{d+2}}$, where $x = (1 + 2\|\widetilde{\boldsymbol{A}}\|_\infty)(1 + \|\boldsymbol{c}''\|_\infty + \|\Delta(u')\|_\infty)$.

Let σ' be the word obtained from u'' by retaining only the actions in $\boldsymbol{A}$. We get $\Delta(u'') = \Delta(\sigma') - i'_0\boldsymbol{p_0} - \cdots - i'_m\boldsymbol{p}_m$ by introducing i'_j, the number of times $-\boldsymbol{p}_j$ occurs in u''. Hence $\Delta(\sigma') = \boldsymbol{p_0} + i'_0\boldsymbol{p_0} + \cdots + i'_m\boldsymbol{p}_m$, since $\Delta(u'') = \boldsymbol{p}_0$.

It follows that $\boldsymbol{Z}'\Delta(\sigma') \geq \boldsymbol{n}'$ and so $\boldsymbol{Z}\Delta(\sigma') \geq \boldsymbol{n}$. Recall that $\boldsymbol{c}' \leq \boldsymbol{c} \leq \boldsymbol{x}$ and that $\boldsymbol{c}' \xrightarrow{-\boldsymbol{p}_0} \boldsymbol{c}'' \xrightarrow{u''} \boldsymbol{c}'$. Since σ' is obtained from u'' by removing some actions in $\widetilde{\boldsymbol{A}} \setminus \boldsymbol{A}$ and since those actions have value 0 in indices where $\boldsymbol{x}$ is not ω, we infer that $\boldsymbol{x} \xrightarrow{-\boldsymbol{p}_0} \boldsymbol{x} \xrightarrow{\sigma'} \boldsymbol{x}$. It remains to bound the length of σ' to conclude the proof. We have $x = (1 + 2\|\widetilde{\boldsymbol{A}}\|_\infty)(1 + \|\boldsymbol{c}''\|_\infty + \|\Delta u'\|_\infty)$ and $|\sigma'| \leq |u''| \leq 17d^2x^{15d^{d+2}}$. After some simplifications, it can be infered that $|\sigma'| \leq (\|\boldsymbol{Z}\|_{1,\infty} + \|\boldsymbol{A}\|_\infty + \|\boldsymbol{n}\|_\infty)^{rc_2^{d^3}}$ for a suitably chosen constant c_2.

The simplification involves calculations that are a bit tedious and can be found in the full version. □

In Lemma 6.4 above, $\boldsymbol{n}$ is a vector of natural numbers in the linear system $\boldsymbol{Z}\Delta(\sigma) \geq \boldsymbol{n}$. It is unlikely that a similar result about short cycles can be proved when $\boldsymbol{n}$ is an integer vector, since that would imply short witnesses for reachability, as shown by the following remark.

Remark 6.5. Let $\boldsymbol{c}_{\text{init}} \xrightarrow{u} \boldsymbol{c}$ be a run in a VAS $\langle \boldsymbol{A}, \boldsymbol{c}_{\text{init}} \rangle$. We associate to every action $\boldsymbol{a}$ the action $\tilde{\boldsymbol{a}} = (\boldsymbol{a}, 0)$ with an extra component equal to zero. We also introduce the set $\tilde{\boldsymbol{A}} = \{\tilde{\boldsymbol{a}} \mid \boldsymbol{a} \in \boldsymbol{A}\} \cup \{(\boldsymbol{c}_{\text{init}}, 2), (-\boldsymbol{c}, 3)\}$. From the word $u = \boldsymbol{a}_1 \dots \boldsymbol{a}_k$ we get the word $\tilde{u} = \tilde{\boldsymbol{a}}_1 \dots \tilde{\boldsymbol{a}}_k$. Now observe that $\sigma = (\boldsymbol{c}_{\text{init}}, 2)\tilde{u}(-\boldsymbol{c}, 3)$ and $\boldsymbol{x} = (\boldsymbol{0}, \omega)$ satisfies $\boldsymbol{x} \xrightarrow{\sigma} \boldsymbol{x}$ and $\Delta(\sigma)(d+1) = 5$, which can be encoded by two inequalities provided that we allow comparisons with negative integers (as we only permit $\geq$). Moreover from any word $\sigma' \in \tilde{\boldsymbol{A}}^*$ such that $\boldsymbol{x} \xrightarrow{\sigma'} \boldsymbol{x}$ and $\Delta(\sigma')(d+1) = 5$ we derive a word $u' \in \boldsymbol{A}^*$ such that $|u'| = |\sigma'| - 2$ and $\boldsymbol{c}_{\text{init}} \xrightarrow{u'} \boldsymbol{c}$. In fact, we observe that u' contains one occurrence of $(\boldsymbol{c}_{\text{init}}, 2)$ and one occurrence of $(-\boldsymbol{c}, 3)$. By removing these occurrences from σ', we get u'. □

7 Small Pumping Segments through Short Cycles

In this section, we use the result of the previous section to prove Theorem 5.2. We first provide a bound on the pumping segments.

Lemma 7.1. *Suppose there is a self-covering sequence $(u_1, \sigma_1, \dots, u_k, \sigma_k)$ satisfying ϕ. Then there is a self-covering sequence $(u'_1, \sigma'_1, \dots, u'_k, \sigma'_k)$ satisfying ϕ such that $|\sigma'_1| + \dots + |\sigma'_k| \leq (\|\boldsymbol{A}\|_\infty + \|\phi\|_1)^{r(\phi)c_1^{(d \cdot k(\phi))^3}}$ where c_1 is a constant.*

Proof (Sketch). A self-covering sequence $(u_1, \sigma_1, \dots, u_k, \sigma_k)$ satisfies a formula $t \geq n$ if, and only if, $\boldsymbol{z} \odot (\Delta(\sigma_1), \dots, \Delta(\sigma_k)) \geq n$, where $\boldsymbol{z} \in \mathbb{Z}^{d \cdot k}$ is an integer vector that only depends on t, $\odot$ is the usual dot product. The conditions $\|\Delta(\sigma_j)\|^- \subseteq \cup_{1 \leq j' < j} \|\Delta(\sigma_{j'})\|^+$ can also be expressed as a set of inequalities of the previous form. By suitably defining a (kd)-dimensional VAS, we can think of a word σ whose displacement is $(\Delta(\sigma_1), \dots, \Delta(\sigma_k))$. The combination of all the satisfied terms of ϕ and the condition for self-covering sequences gives rise to a linear system $\boldsymbol{Z}\Delta(\sigma) \geq \boldsymbol{n}$, where $\boldsymbol{Z}$ is an integer matrix and $\boldsymbol{n}$ is a vector of natural numbers.

Using the result of the previous section, we find a short cycle labeled by σ' whose displacement also satisfies $\boldsymbol{Z}\Delta(\sigma') \geq \boldsymbol{n}$. From this short cycle, we can extract words of the original VAS which are pumping segments satisfying the length requirements of the lemma. □

We now have the necessary ingredients to prove our main result, Theorem 5.2. Assume that $\langle \boldsymbol{A}, \boldsymbol{c}_{\text{init}} \rangle$ admits a self-covering sequence $(u_1, \sigma_1, \dots, u_k, \sigma_k)$ satisfying ϕ. By Lemma 7.1, there exists a self-covering sequence $(u'_1, \sigma'_1, \dots, u'_k, \sigma'_k)$ satisfying ϕ such that $|\sigma'_1| + \dots + |\sigma'_k| \leq (\|\boldsymbol{A}\|_\infty + \|\phi\|_1)^{r(\phi)c_1^{(k(\phi) \cdot d)^3}}$. We derive

from Lemma 5.4 that there exists a self-covering sequence $(u''_1, \sigma'_1, \ldots, u''_k, \sigma'_k)$ such that $|u''_1|+\cdots+|u''_k| \leq (2\|\boldsymbol{A}\|_\infty(\|\boldsymbol{A}\|_\infty+\|\phi\|_1)^{r(\phi)c_1^{(k(\phi)\cdot d)^3}})^{((d+1)k(\phi)+1)!}$. Simplifying this, we get $|\sigma'_1|+\cdots+|\sigma'_k|+|u''_1|+\cdots+|u''_k| \leq (\|\boldsymbol{A}\|_\infty+\|\phi\|_1)^{r(\phi)c^{(k(\phi)\cdot d)^3}}$ for a suitably chosen constant c, which concludes the proof of the theorem.

We now define the size of a VAS and a formula and state a complexity theoretic consequence of the small model property obtained above. The *size* of a VAS is the obvious one, where integers are encoded in binary. The *size* $|t|$ of a term t is defined inductively as follows: $|z\boldsymbol{\delta}_j(i)| = \log(|z|+1)$ and $|t_1+t_2| = 1+|t_1|+|t_2|$. The *size* $|\phi|$ of a formula ϕ is defined by $|t \geq n| = |t|+1+\log(n+1)$, $|\phi_1 \vee \phi_2| = |\phi_1|+1+|\phi_2|$ and $|\phi_1 \wedge \phi_2| = |\phi_1|+1+|\phi_2|$.

Corollary 7.2. *Given a VAS $\langle \boldsymbol{A}, \boldsymbol{c}_{init}\rangle$ and a formula ϕ, the problem of checking whether there is a self-covering sequence satisfying ϕ is* ExpSpace*-complete.*

Proof. For the exponential space lower bound, we have seen in Section 4 that we can reduce the boundedness problem to checking a formula of our logic. Since the boundedness problem is ExpSpace-hard [9], checking whether a given formula is satisfied by a given VAS is ExpSpace-hard.

For the exponential space upper bound, a non-deterministic Turing machine can guess and verify the existence of a self-covering sequence of length at most $(\|\boldsymbol{A}\|_\infty + \|\phi\|_1)^{r(\phi)c^{(d\cdot k(\phi))^3}}$. The Turing machine needs to maintain a counter to count (in binary) up to a maximum of $(\|\boldsymbol{A}\|_\infty + \|\phi\|_1)^{r(\phi)c^{(d\cdot k(\phi))^3}}$ and store at most $2k$ intermediate configurations. The memory requirement is therefore $\mathcal{O}(r(\phi)c^{(d\cdot k(\phi))^3}(\log\|\boldsymbol{A}\|_\infty + \log\|\phi\|_1))$. It is easy to see that the size of the VAS is an upper bound on $\log\|\boldsymbol{A}\|_\infty$ and the size $|\phi|$ of the formula ϕ is an upper bound on $\log\|\phi\|_1$. Hence, the well-known Savitch's theorem then gives a deterministic Turing machine that works in exponential space. □

8 Complexity of the Context-Freeness Problem for VAS

We have shown in the previous sections that the model-checking problem for our logic can be solved in exponential space. As an application, we now focus on the context-freeness problem for VAS, and characterize its complexity.

The context-freeness problem asks whether the trace language of a given VAS is context-free. This problem was shown to be decidable by Schwer in [14]. Since it is based on the coverability graph, the resulting algorithm's complexity is non-primitive recursive. Recently, we revisited the context-freeness problem for VAS, and gave a simpler proof of decidability [8]. Our approach is based on regular bounded languages having a non-context free intersection with the set of traces. In this section, we briefly recall this characterization. Then, we show how to express it by a formula in our logic, thereby providing an exponential space upper bound for the context-freeness problem.

A pair $(\boldsymbol{v}_1, \boldsymbol{v}_2)$ of vectors in $\mathbb{Z}^d$ such that $\boldsymbol{v}_1 \geq \boldsymbol{0}$ and $\boldsymbol{v}_2 \not\geq \boldsymbol{0}$ is called a *matching pair*. For every matching pair $(\boldsymbol{v}_1, \boldsymbol{v}_2)$, there exists a maximal non-negative rational number $\lambda \geq 0$ such that $\boldsymbol{v}_1 + \lambda\boldsymbol{v}_2 \geq \boldsymbol{0}$. We call this rational

number the *ratio* of the matching pair $(\boldsymbol{v}_1, \boldsymbol{v}_2)$, and we denote it by $\mathsf{rat}(\boldsymbol{v}_1, \boldsymbol{v}_2)$. We define the *excess* of $(\boldsymbol{v}_1, \boldsymbol{v}_2)$ as the vector $\mathsf{exc}(\boldsymbol{v}_1, \boldsymbol{v}_2) = \boldsymbol{v}_1 + \mathsf{rat}(\boldsymbol{v}_1, \boldsymbol{v}_2) \cdot \boldsymbol{v}_2$. Note that $\mathsf{exc}(\boldsymbol{v}_1, \boldsymbol{v}_2) \geq \mathbf{0}$.

A *matching scheme* is a tuple $(\sigma_1, \ldots, \sigma_k, U)$ where $\sigma_1, \ldots, \sigma_k$ are words in $(\mathbb{Z}^d)^*$ and U is a nested binary relation on $\{1, \ldots, k\}$ such that $(\Delta(\sigma_s), \Delta(\sigma_t))$ is a matching pair for every $(s,t) \in U$. Here, by *nested*, we mean that U satisfies the two following conditions:

$$(s,t) \in U \;\Rightarrow\; s \leq t \tag{1}$$

$$(r,t) \in U \wedge (s,u) \in U \;\Rightarrow\; \neg(r < s < t < u) \tag{2}$$

The *excess* of a matching scheme $(\sigma_1, \ldots, \sigma_k, U)$ is the vector $\mathsf{exc}(\sigma_1, \ldots, \sigma_k, U) = \sum_{(s,t)\in U} \mathsf{exc}(\Delta(\sigma_s), \Delta(\sigma_t))$.

Definition 8.1. *A* witness of non-context-freeness *for a VAS $\langle \boldsymbol{A}, \boldsymbol{c}_{init} \rangle$ is a tuple $(u_1, \sigma_1, \ldots, u_k, \sigma_k, U)$, where u_i, σ_i are words in $\boldsymbol{A}^*$ and $(\sigma_1, \ldots, \sigma_k, U)$ is a matching scheme, such that:*

1. *The word $u_1\sigma_1 \cdots u_k\sigma_k$ is a trace of $\langle \boldsymbol{A}, \boldsymbol{c}_{init} \rangle$,*
2. *It holds that $\Delta(\sigma_k) \not\geq \mathbf{0}$ and $\|\Delta(\sigma_k)\|^- \subseteq \|\mathsf{exc}(\sigma_1, \ldots, \sigma_k, U)\|^+$, and*
3. *For every $(s,t) \in U$ with $t < k$, there exists $(r,t) \in U$ such that $r \leq s$ and $\|\Delta(\sigma_t)\|^- \subseteq \|\Delta(\sigma_r)\|^+$.*

Theorem 8.2 ([8]). *The trace language of a VAS $\langle \boldsymbol{A}, \boldsymbol{c}_{init} \rangle$ is not context-free if, and only if, $\langle \boldsymbol{A}, \boldsymbol{c}_{init} \rangle$ admits a witness of non-context-freeness.*

Our objective is to express non-context-freeness by a formula in our relational trace logic. However, the conditions of Definition 8.1 cannot be translated, as is, in the logic. Firstly, the number k of pumping segments is not, a priori, bounded. Secondly, the sequence $(u_1, \sigma_1, \ldots, u_k, \sigma_k)$ need not be a self-covering sequence. Lastly, membership of a given index in the set $\|\mathsf{exc}(\sigma_1, \ldots, \sigma_k, U)\|^+$ is not linear[1] in $\Delta(\sigma_1), \ldots, \Delta(\sigma_k)$ since it requires comparing ratios between components. To overcome this difficulty, we show that it is enough to look for witnesses of non-context-freeness satisfying additional, simplifying requirements.

Formally, a witness of non-context-freeness $(u_1, \sigma_1, \ldots, u_k, \sigma_k, U)$ is called *perfect* if $k \leq 3d + 1$, the tuple $(u_1, \sigma_1, \ldots, u_k, \sigma_k)$ is a self-covering sequence, and $\mathsf{rat}(\Delta(\sigma_s), \Delta(\sigma_t)) \in \{0, 1\}$ for every $(s,t) \in U$.

Proposition 8.3. *The trace language of a VAS $\langle \boldsymbol{A}, \boldsymbol{c}_{init} \rangle$ is not context-free if, and only if, $\langle \boldsymbol{A}, \boldsymbol{c}_{init} \rangle$ admits a perfect witness of non-context-freeness.*

Proof (Sketch). We show that every witness of non-context-freeness can be transformed into a perfect one. The proposition then follows from Theorem 8.2. Consider a witness of non-context-freeness $(u_1, \sigma_1, \ldots, u_k, \sigma_k, U)$, and assume, w.l.o.g., that U is minimal with respect to inclusion. We can show that U contains

[1] Given a matching pair $(\boldsymbol{v}_1, \boldsymbol{v}_2)$ and an index i such that $\boldsymbol{v}_2(i) \leq 0$, it holds that $i \in \|\mathsf{exc}(\boldsymbol{v}_1, \boldsymbol{v}_2)\|^+$ if, and only if, $\boldsymbol{v}_1(i) \cdot \boldsymbol{v}_2(j) < \boldsymbol{v}_1(j) \cdot \boldsymbol{v}_2(i)$ for some index $j \neq i$.

at most two pairs for each index $i \in \|\mathsf{exc}(\sigma_1, \ldots, \sigma_k, U)\|^+$, hence, the support $S = \{s, t \mid (s,t) \in U\}$ of U has a cardinality of at most $3d$. Obviously, we may transform the witness by keeping only the pumping segments σ_i for $i \in S \cup \{k\}$. The remaining pumping segments are merged together with the words u_i that surround them. By construction, the resulting witness of non-context-freeness is a self-covering sequence with at most $3d+1$ pumping segments since $|S| \leq 3d$. It remains to enforce the ratios to be in $\{0, 1\}$. Pick a pair $(s,t) \in U$ such that $\mathsf{rat}(\Delta(\sigma_s), \Delta(\sigma_t))$ is a positive rational number, written $\frac{p}{q}$. Observe that

$$\begin{aligned} \mathsf{rat}(n_1 \boldsymbol{v}_1, n_2 \boldsymbol{v}_2) &= \frac{n_1}{n_2} \cdot \mathsf{rat}(\boldsymbol{v}_1, \boldsymbol{v}_2) \\ \mathsf{exc}(n_1 \boldsymbol{v}_1, n_2 \boldsymbol{v}_2) &= n_1 \cdot \mathsf{exc}(\boldsymbol{v}_1, \boldsymbol{v}_2) \end{aligned}$$

for every matching pair $(\boldsymbol{v}_1, \boldsymbol{v}_2)$ and positive natural numbers n_1 and n_2. So we define $\sigma'_s = \sigma_s^q$, $\sigma'_t = \sigma_t^p$ and $\sigma'_i = \sigma_i^n$ for $i \notin \{s,t\}$, where n is equal to p or q. We derive from Lemma 2.2 that there exists $u'_1, \ldots, u'_k$ such that $(u'_1, \sigma'_1, \ldots, u'_k, \sigma'_k, U)$ is a witness of non-context-freeness. This transformation guarantees that $\mathsf{rat}(\Delta(\sigma'_s), \Delta(\sigma'_t)) = 1$, however, it may also change the ratios of other pairs involving s (if $n = p$) or t (if $n = q$). Still, as $(\sigma_1, \ldots, \sigma_k, U)$ is a matching scheme, it is possible to process the pairs $(s,t) \in U$ in an appropriate order that prevents such conflicts. □

Example 8.4. Consider the VAS $\langle \boldsymbol{A}, \boldsymbol{c}_{\text{init}} \rangle$ given by $\boldsymbol{A} = \{\boldsymbol{a}, \boldsymbol{b}\}$ and $\boldsymbol{c}_{\text{init}} = (2,2)$, where $\boldsymbol{a} = (-2,3)$ and $\boldsymbol{b} = (3,-2)$. The tuple $(\varepsilon, \boldsymbol{ab}, \varepsilon, \boldsymbol{a}, \varepsilon, \boldsymbol{b}, U)$, with $U = \{(1,2),(1,3)\}$, is a witness of non-context-freeness. This witness is not perfect since $\mathsf{rat}(\Delta(\boldsymbol{ab}), \Delta(\boldsymbol{a})) = \mathsf{rat}((1,1),(-2,3)) = \frac{1}{2}$. Replacing $\boldsymbol{ab}$ by $\boldsymbol{abab}$ in the witness makes it perfect. □

We now explain how to encode by a formula in our logic the conditions of perfect witnesses of non-context-freeness. Consider a positive natural number k and a nested relation U on $\{1, \ldots, k\}$. Firstly, we express that $(\Delta(\sigma_s), \Delta(\sigma_t))$ is a matching pair with ratio in $\{0,1\}$ for every $(s,t) \in U$, by the following formula:

$$\bigwedge_{(s,t) \in U} \left(\bigwedge_{i=1}^{d} \boldsymbol{\delta}_s(i) \geq 0 \ \wedge \ \bigvee_{i=1}^{d} -\boldsymbol{\delta}_t(i) \geq 1 \ \wedge \ (\rho_0(s,t) \vee \rho_1(s,t)) \right)$$

where $\rho_0(s,t)$ and $\rho_1(s,t)$ are formulas, expressible in our logic, specifying that the matching pair $(\Delta(\sigma_s), \Delta(\sigma_t))$ has ratio 0 and 1, respectively.
Secondly, we encode the requirements of Definition 8.1. The condition that $\Delta(\sigma_k) \not\geq \mathbf{0}$ is expressed by the formula $\bigvee_{i=1}^{d} -\boldsymbol{\delta}_k(i) \geq 1$. For the encoding of the condition $\|\Delta(\sigma_k)\|^- \subseteq \|\mathsf{exc}(\sigma_1, \ldots, \sigma_k, U)\|^+$, we exploit the property that the ratio of each matching pair $(\Delta(\sigma_s), \Delta(\sigma_t))$ is either 0 or 1, as follows:

$$\bigwedge_{i=1}^{d} \left(\boldsymbol{\delta}_k(i) \geq 0 \ \vee \bigvee_{(s,t) \in U} (\rho_0(s,t) \ \wedge \ \boldsymbol{\delta}_s(i) \geq 1) \vee (\rho_1(s,t) \ \wedge \ \boldsymbol{\delta}_s(i) + \boldsymbol{\delta}_t(i) \geq 1) \right)$$

The last condition of Definition 8.1 is expressed by the following formula:

$$\bigwedge_{(s,t)\in U, t<k} \left(\bigvee_{(r,t)\in U, r\leq s} \bigwedge_{i=1}^{d} \boldsymbol{\delta}_t(i) \geq 0 \ \vee \ \boldsymbol{\delta}_r(i) \geq 1 \right)$$

Let $\varphi(k, U)$ be the conjunction of the above formulas, and let ψ denote the disjunction of all $\varphi(k, U)$ where $1 \leq k \leq 3d + 1$ and U is a nested relation on $\{1, \ldots, k\}$. As intended, the formula ψ expresses non-context-freeness of the trace language. We derive the complexity of the context-freeness problem for VAS from the analysis of our logic developed in the previous sections.

Theorem 8.5. *The context-freeness problem for VAS is* ExpSpace*-complete.*

Proof. By construction, $\langle \boldsymbol{A}, \boldsymbol{c}_{\text{init}} \rangle$ admits a perfect witness of non-context-freeness if, and only if, $\langle \boldsymbol{A}, \boldsymbol{c}_{\text{init}} \rangle$ satisfies ψ. It follows from Proposition 8.3 that the trace language of $\langle \boldsymbol{A}, \boldsymbol{c}_{\text{init}} \rangle$ is not context-free if, and only if, $\langle \boldsymbol{A}, \boldsymbol{c}_{\text{init}} \rangle$ satisfies ψ. It is readily seen that $|\psi|$ is at most exponential in the dimension d, and that the conjunction rank of ψ is bounded by a polynomial in d. We derive from Theorem 5.2, with the same arguments as in Corollary 7.2, that the context-freeness problem for VAS can be solved in exponential space. The exponential space lower bound is obtained by a reduction from the boundedness problem for VAS. □

9 Discussion and Future Work

We introduced a logic that can express positive Boolean combinations of linear relations among the displacements of pumping segments in self-covering sequences. We showed that if a VAS satisfies a formula, there are witnessing self-covering sequences whose size is at most doubly-exponential in the size of the VAS and the formula. This gives an exponential space upper bound for the model-checking problem for our logic, which in turn gives an exponential space upper bound for the context-freeness problem.

Yen introduced a logic similar to ours in [17], interpreted over all traces instead of self-covering sequences like we do. Atig and Habermehl showed in [1] that the problem of checking whether there is a trace satisfying a given formula in Yen's path logic is in the class $\mathcal{C}_{\text{RP}}$ (i.e., equivalent to the reachability problem). They also give a fragment of Yen's path logic that can be checked in exponential space. This fragment imposes the condition that the total displacement of the words under consideration is greater than or equal to $\mathbf{0}$, which is incomparable with our restriction to self-covering sequences and also incompatible with witnesses of non-context-freeness.

A logic similar to ours was introduced by Demri in [3], interpreted over self-covering sequences. In place of $t \geq n$ in our logic, the conditions allowed in [3] can constrain a variable to be inside any interval of integers. However, we allow combining $\Delta(\sigma_1), \ldots, \Delta(\sigma_k)$ in a single term, which is not allowed in [3].

Hence, the two logics are incomparable. The inability of the logic in [3] to combine $\Delta(\sigma_1), \ldots, \Delta(\sigma_k)$ in a single term renders it unable to express the presence of witnesses of non-context-freeness.

Blockelet and Schmitz introduced in [2] a fragment of computational tree logic enriched with formulas in Presburger arithmetic for expressing properties of coverability graphs. An exponential space upper bound is provided for a fragment of this logic by imposing a so-called *eventually increasing* condition that is similar to the one imposed in [1], but for trees instead of paths. Again, this condition is incompatible with witnesses of non-context-freeness.

It will be interesting to see if the techniques used in the above collection of incomparable logics can be unified to define a logic that extends all of them and that can still be checked in exponential space.

References

1. Atig, M.F., Habermehl, P.: On Yen's path logic for Petri nets. Int. J. Found. Comput. Sci. 22(4), 783–799 (2011)
2. Blockelet, M., Schmitz, S.: Model checking coverability graphs of vector addition systems. In: Murlak, F., Sankowski, P. (eds.) MFCS 2011. LNCS, vol. 6907, pp. 108–119. Springer, Heidelberg (2011)
3. Demri, S.: On selective unboundedness of VASS. In: *Proc. INFINITY*. EPTCS, vol. 39, pp. 1–15 (2010)
4. Esparza, J., Nielsen, M.: Decidability issues for Petri nets - a survey. Bulletin of the EATCS 52, 244–262 (1994)
5. Karp, R.M., Miller, R.E.: Parallel program schemata. Journal of Computer and System Sciences 3(2), 147–195 (1969)
6. Kosaraju, S.R.: Decidability of reachability in vector addition systems. In: Proc. STOC, pp. 267–281. ACM (1982)
7. Leroux, J.: Vector addition system reversible reachability problem. Logical Methods in Computer Science 9(1) (2013)
8. Leroux, J., Penelle, V., Sutre, G.: On the context-freeness problem for vector addition systems. In: Proc. LICS. IEEE (to appear, 2013)
9. Lipton, R.J.: The reachability problem requires exponential space. Technical Report 62, Yale University (1976)
10. Mayr, E.W.: An algorithm for the general Petri net reachability problem. In: Proc. STOC, pp. 238–246. ACM (1981)
11. Mayr, E.W., Meyer, A.R.: The complexity of the finite containment problem for Petri nets. J. ACM 28(3), 561–576 (1981)
12. 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)
13. Rackoff, C.: The covering and boundedness problems for vector addition systems. Theoretical Computer Science 6(2), 223–231 (1978)
14. Schwer, S.R.: The context-freeness of the languages associated with vector addition systems is decidable. Theor. Comput. Sci. 98(2), 199–247 (1992)
15. Valk, R., Jantzen, M.: The residue of vector sets with applications to decidability problems in Petri nets. Acta Inf. 21, 643–674 (1985)
16. Valk, R., Vidal-Naquet, G.: Petri nets and regular languages. Journal of Computer and System Sciences 23(3), 299–325 (1981)
17. Yen, H.-C.: A unified approach for deciding the existence of certain Petri net paths. Inf. Comput. 96(1), 119–137 (1992)

Expand, Enlarge, and Check for Branching Vector Addition Systems

Rupak Majumdar and Zilong Wang

MPI-SWS, Germany

Abstract. Expand, enlarge, and check (EEC) is a successful heuristic for the coverability problem of well-structured transition systems. EEC constructs a sequence of under- and over-approximations with the property that the presence of a bug is eventually exhibited by some under-approximation and the absence of a bug is eventually exhibited by some over-approximation.

In this paper, we consider the application of EEC to the coverability problem for branching vector addition systems (BVAS), an expressive model that subsumes Petri nets. We describe an EEC algorithm for BVAS, and prove its termination and correctness. We prove an upper bound on the number of iterations for our EEC algorithm, both for BVAS and, as a special case, vector addition systems (or Petri nets). We show that in addition to practical effectiveness, the EEC heuristic is asymptotically optimal. For BVAS, it requires at most doubly-exponentially many iterations, thus matching the optimal 2EXPTIME upper bound. For Petri nets, it can be implemented in EXPSPACE, again matching the optimal bound. We have implemented our algorithm and used it to verify safety properties of concurrent programs with asynchronous tasks.

1 Introduction

Branching vector addition systems (BVAS) are an expressive model that generalize vector addition systems (VAS, or Petri nets) with branching structures. Intuitively, one can consider a VAS as producing a linear sequence of vectors using unary rewrite rules, where a rewrite rule takes a vector v and adds a constant δ to it, as long as the sum $v + \delta$ remains non-negative on all co-ordinates. A branching VAS adds a second, binary rewrite rule that takes two vectors v_1 and v_2 and rewrites them to $v_1 + v_2 + \delta$ for a constant δ, again provided the sum is non-negative on all co-ordinates. Thus, a BVAS generates a derivation tree of vectors, starting with a multiset of initial vectors, or axioms, at the leaves and generating a vector at the root of a derivation, where each internal node in the tree applies a unary or a binary rewrite rule. The reachability problem for BVAS is to check if a given vector can be derived, and the coverability problem asks, given a vector v, if a vector $v' \geq v$ can be derived. These generalize the corresponding problems for VAS. Several verification problems, such as the analysis of recursively parallel programs [1] and the analysis of some cryptographic protocols [17], have been shown to reduce to the coverability problem for BVAS.

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 152–166, 2013.

Coverability for BVAS is known to be decidable, both through a generalized Karp-Miller construction [16] as well as through a bounding argument [5]. Further, the bounding argument characterizes the complexity of the problem: coverability is 2EXPTIME-complete [5] (contrast with the EXPSPACE-completeness for VAS [15]). The Karp-Miller construction is non-primitive recursive, since BVAS subsume VAS [11].

Despite potential applications, the study of BVAS has so far remained in the domain of theoretical results, and to the best of our knowledge, there have not been any attempts to build analysis tools for coverability. In contrast, tools for VAS coverability have made steady progress and can now handle quite large benchmarks derived from the analysis of multi-threaded programs [10,13]. In our view, one reason is that a direct implementation of the algorithms from [16,5] are unlikely to perform well: Karp-Miller trees for VAS do not perform well in practice, and Demri et al.'s complexity-theoretically optimal algorithm performs a non-deterministic guess and enumeration by an alternating Turing machine.

In this paper, we apply the *expand, enlarge, and check* paradigm (EEC) [7] to the analysis of BVAS. EEC is a successful heuristic for checking coverability of well-structured transition systems such as Petri nets. It constructs a sequence of under- and over-approximations of the state space of a system such that, for a target state t, (1) if t is coverable, then a witness is found by an under-approximation, (2) if t is not coverable, then a witness for un-coverability is found by an over-approximation, and (3) eventually, one of the two outcomes occur and the algorithm terminates.

EEC offers several nice features for implementation. First, each approximation it considers is finite-state, thus opening the possibility of applying model checkers for finite-state systems. Second, EEC is goal-directed: it computes abstractions that are precise enough to prove or disprove coverability of a target, unlike a Karp-Miller procedure that computes the exact coverability set independent of the target. Third, it allows a forward abstract exploration of the state space, which is often more effective in practice.

Our first contribution is to port the EEC paradigm to the coverability analysis of BVAS. We show how to construct a sequence of under- and over-approximations of derivations such that if a target is coverable, an under-approximation derives a witness for coverability, and if a target is not coverable, an over-approximation derives a witness for un-coverability. We generalize the proof of correctness of EEC for well-structured systems. Since there is no BVAS analogue of a backward-reachability algorithm for VAS, our proofs instead use induction on derivations and the Karp-Miller construction of [16].

A natural question is how well EEC performs in the worst case compared to asymptotically optimal algorithms. For example, even for VAS, it is unknown if the EEC algorithm can match the known EXPSPACE upper bound for coverability, or if it matches the non-primitive recursive lower bound for Karp-Miller trees. Our second contribution is to bound the number of iterations of the EEC algorithm in the worst case. We show that we can compute a constant c of size doubly exponential in the size of the BVAS and the target vector such that the

EEC algorithm is guaranteed to terminate in c iterations. In each iteration, the algorithm explores approximate state spaces of derivations, that correspond to exploring AND-OR trees of size doubly exponential in the input. In other words, if each exploration is performed optimally, we get an optimal asymptotic upper bound for EEC. Specifically, for VAS, we get an EXPSPACE upper bound, since there are doubly exponential iterations and each iteration checks two reachability problems over doubly-exponential state spaces. (In practice though, model checkers do not implement space-optimal reachability procedures.) While our proof uses Rackoff-style bounds [15,5], our implementation does not require any knowledge of these bounds. A similar argument was used in [2] to show a doubly exponential bound on the backward reachability algorithm for VAS.

We have implemented the EEC-based procedure for BVAS coverability. Our motivation for analyzing BVAS came from the analysis of recursively parallel programs [6,1]. It is known that the analysis of asynchronous programs, a cooperatively scheduled concurrency model, can be reduced to coverability of VAS [6], and there have been EEC-based tools for these programs [9]. However, some asynchronous programs use features such as posting a set of tasks in a handler and waiting on the first task to return, that are not reducible to asynchronous programs. Bouajjani and Emmi [1] define a class of recursively parallel programs that can express such constructs, and show that the safety verification problem for this class is equivalent to coverability of BVAS. We applied this reduction in our implementation, and used our tool to model check safety properties of recursively parallel programs. We coded the control flow of tasks in a simple web server [4] and showed that our tool can successfully check for safety properties and find bugs. On our examples, the EEC algorithm terminates in one iteration, that is, with a $\{0, 1, \infty\}$ abstraction. While our evaluations are preliminary, we believe there is a potential for model checking tools for complex concurrent programs based on BVAS coverability.

2 Preliminaries

Well Quasi Ordering. A *quasi ordering* $(X, \preceq)$ is a reflexive and transitive binary relation on X. A quasi ordering $(X, \preceq)$ is a *well quasi ordering* iff for every infinite sequence $x_0, x_1, \ldots$ of elements from X, there exists $i < j$ with $x_i \preceq x_j$. A subset X' of X is *upward closed* if for each $x \in X$, if there is an $x' \in X'$ with $x' \preceq x$ then $x \in X'$. A subset X' of X is *downward closed* if for each $x \in X$, if there is an $x' \in X'$ with $x \preceq x'$ then $x \in X'$. Given $x \in X$, we write $x{\downarrow}$ and $x{\uparrow}$ for the *downward closure* $\{x' \in X \mid x' \preceq x\}$ and *upward closure* $\{x' \in X \mid x \preceq x'\}$ of x respectively. Downward and upward closures are naturally extended to sets, i.e., $X{\downarrow} = \bigcup_{x \in X} x{\downarrow}$ and $X{\uparrow} = \bigcup_{x \in X} x{\uparrow}$. A subset $S \subseteq X$ is *minimal* iff for every two elements $x, x' \in S$, we have $x \not\preceq x'$.

Numbers and Vectors. We write $\mathbb{N}$, $\mathbb{N}^+$ and $\mathbb{Z}$ for the set of non-negative, positive and arbitrary integers, respectively. Given two integers a and b, we write $[a, b]$ for $\{n \in \mathbb{Z} \mid a \leq n \leq b\}$.

For a vector $v \in \mathbb{Z}^k$ and $i \in [1, k]$, we write $v[i]$ for the ith component of v. Given two vectors $v, v' \in \mathbb{Z}^k$, $v \leq v'$ iff for all $i \in [1, k]$, $v[i] \leq v'[i]$. Moreover,

$v < v'$ iff $v \leq v'$ and $v' \not\leq v$. It is well-known that $(\mathbb{N}^k, \leq)$ is a well quasi ordering. We write $\mathbf{0}$ for the zero vector.

Given a finite set $S \subseteq \mathbb{Z}$ of integers, we write $\max(S)$ for the greatest integer in the set. We define $\max(\emptyset) = 0$. Given a vector $v \in \mathbb{Z}^k$, let $\max(v) = \max(\{v[1], \ldots, v[k]\})$. When $k = 0$, we have $\max(\langle\rangle) = 0$. We define $\min(S)$ analogously. We write $\min(0, v)$ for the vector $\langle \min(\{0, v[1]\}), \ldots, \min(\{0, v[k]\})\rangle$. The vector $\max(0, v)$ is defined analogously. For simplicity, we write v^- for the vector $-\min(0, v)$ and v^+ for the vector $\max(0, v)$. Given a finite set of vectors $R \subseteq \mathbb{Z}^k$, let $R^{-/+}$ be the set $\{v^{-/+} \mid v \in R\}$ respectively. We define $\max(R) = \max(\{\max(v^+) \mid v \in R\})$. The size of a vector is the number of bits required to encode it, all numbers being encoded in binary.

Trees. A *finite binary tree* $\mathcal{T}$, which may contain nodes with one child, is a non-empty finite subset of $\{1, 2\}^*$ such that, for all $n \in \{1, 2\}^*$ and $i \in \{1, 2\}$, $n \cdot 2 \in \mathcal{T}$ implies $n \cdot 1 \in \mathcal{T}$, and $n \cdot i \in \mathcal{T}$ implies $n \in \mathcal{T}$. The nodes of $\mathcal{T}$ are its elements. The root of $\mathcal{T}$ is ε, the empty word. All notions such as parent, child, subtree and leaf, have their standard meanings. The height of $\mathcal{T}$ is the number of nodes in the longest path from the root to a leaf.

BVAS, Derivations, and Coverability. A *branching vector addition system* (BVAS) [16,5] is a tuple $\mathcal{B} = \langle k, A, R_1, R_2\rangle$, where $k \in \mathbb{N}$ is the *dimension*, $A \subseteq \mathbb{N}^k$ is a non-empty finite set of *axioms*, and $R_1, R_2 \subseteq \mathbb{Z}^k$ are finite sets of *unary and binary rules*, respectively. The size of a BVAS, *size*$(\mathcal{B})$ is the number of bits required to encode a BVAS, where numbers are encoded in binary.

The semantics of a BVAS $\mathcal{B}$ is captured using derivations. Intuitively, a derivation starts with a number of axioms from A, proceeds by applying rules from $R_1 \cup R_2$, and ends with a single vector. Applying a unary rule means adding it to a derived vector, and applying a binary rule means adding it to the sum of two derived vectors. While applying rules, all derived vectors are required to be non-negative. Formally, a *derivation* $\mathcal{D}$ of $\mathcal{B}$ is defined inductively as follows.

D1: If $v \in A$, then $\overline{v}$ is a derivation.

D2: If $\mathcal{D}_1$ is a derivation with a derived vector $v_1 \in \mathbb{N}^k$, then for each unary rule $\delta_1 \in R_1$ with $\mathbf{0} \leq v_1 + \delta_1$,

$$\mathcal{D} : \dfrac{\begin{array}{c}\vdots\,\mathcal{D}_1\\ v_1\end{array}}{v}\,\delta_1$$

is a derivation, where $v = v_1 + \delta_1$.

D3: If $\mathcal{D}_1$ and $\mathcal{D}_2$ are derivations with derived vectors $v_1, v_2 \in \mathbb{N}^k$ respectively, then for each binary rule $\delta_2 \in R_2$ with $\mathbf{0} \leq v_1 + v_2 + \delta_2$,

$$\mathcal{D} : \dfrac{\begin{array}{cc}\vdots\,\mathcal{D}_1 & \vdots\,\mathcal{D}_2\\ v_1 & v_2\end{array}}{v}\,\delta_2$$

is a derivation, where $v = v_1 + v_2 + \delta_2$.

A derivation $\mathcal{D}$ can be represented as a finite binary tree whose nodes are labelled by non-negative vectors. Therefore, all notions of trees can be naturally applied to derivations. For a derivation $\mathcal{D}$ and its node n, we write $\mathcal{D}(n)$ for the non-negative vector labelled at n. We say $\mathcal{D}$ *derives a vector* v iff $\mathcal{D}(\varepsilon) = v$.

A derivation $\mathcal{D}$ is *compact* iff for each node n and for each its ancestor n', we have $\mathcal{D}(n) \neq \mathcal{D}(n')$. Given a derivation $\mathcal{D}$ with a node n and an ancestor n' of n with $\mathcal{D}(n) = \mathcal{D}(n')$, a *contraction* $\mathcal{D}[n' \leftarrow n]$ over $\mathcal{D}$ is obtained by replacing the subtree rooted at n' with the subtree rooted at n in $\mathcal{D}$. We write $\mathsf{compact}(\mathcal{D})$ for the compact derivation computed by a finite sequence of contractions over $\mathcal{D}$.

Given a BVAS $\mathcal{B} = \langle k, A, R_1, R_2 \rangle$, we say a vector v is *reachable* in $\mathcal{B}$ iff there is a derivation $\mathcal{D}$ with $\mathcal{D}(\varepsilon) = v$. We write $\mathit{Reach}(\mathcal{B}) = \{v \mid \exists \mathcal{D}.\ \mathcal{D}(\varepsilon) = v\}$ for the set of reachable vectors in $\mathcal{B}$. We say a vector v is *coverable* in $\mathcal{B}$ iff there is a derivation $\mathcal{D}$ with $v \leq \mathcal{D}(\varepsilon)$. We call a derivation $\mathcal{D}$ a *covering witness* of v iff $v \leq \mathcal{D}(\varepsilon)$. The *coverability problem* asks, given a BVAS $\mathcal{B}$ and a vector $t \in \mathbb{N}^k$, whether t is coverable in $\mathcal{B}$. Equivalently, t is coverable iff $t \in \mathit{Reach}(\mathcal{B})\downarrow$.

3 Under-and Over-Approximation

We give two approximate analyses for BVAS: an under-approximation that fixes a finite set of vectors and only considers those vectors in that finite set, and an over-approximation that introduces limit elements. The under-approximation can show that a vector is coverable and the over-approximation can prove that a vector is not coverable.

3.1 Underapproximation

Truncated Derivations. Given a BVAS $\mathcal{B} = \langle k, A, R_1, R_2 \rangle$ and an $i \in \mathbb{N}$, define $C_i \subseteq \mathbb{N}^k$ as $A \cup \{0, \ldots, i\}^k$. Given a vector $v \in \mathbb{N}^k$ and an $i \in \mathbb{N}$, We write $\mathsf{under}(v, i)$ for a *truncated vector* such that for all $j \in [1, k]$, $\mathsf{under}(v, i)[j] = v[j]$ if $v[j] \leq i$, $\mathsf{under}(v, i)[j] = i$ otherwise. For all vector $v \in \mathbb{N}^k$ and for all $i \in \mathbb{N}$, $\mathsf{under}(v, i) \leq v$. A *truncated derivation* $\mathcal{F}$ w.r.t. i is defined inductively as follows.

T1: If $v \in A$, then $\overline{v}$ is a truncated derivation.

T2: If $\mathcal{F}_1$ is a truncated derivation with a derived truncated vector $v_1 \in \mathbb{N}^k$, then for each unary rule $\delta_1 \in R_1$ with $\mathbf{0} \leq v_1 + \delta_1$,

$$\mathcal{F} : \dfrac{\begin{array}{c}\vdots \mathcal{F}_1 \\ v_1\end{array}}{v}\,\delta_1$$

is a truncated derivation, where $v = \mathsf{under}(v_1 + \delta_1, i)$.

T3: If $\mathcal{F}_1$ and $\mathcal{F}_2$ are truncated derivations with derived truncated vectors $v_1, v_2 \in \mathbb{N}^k$ respectively, then for each binary rule $\delta_2 \in R_2$ with $\mathbf{0} \leq v_1 + v_2 + \delta_2$,

$$\mathcal{F} : \dfrac{\begin{array}{cc}\vdots \mathcal{F}_1 & \vdots \mathcal{F}_2 \\ v_1 & v_2\end{array}}{v}\,\delta_2$$

is a truncated derivation, where $v = \mathsf{under}(v_1 + v_2 + \delta_2, i)$.

Analogously to derivations, a truncated derivation $\mathcal{F}$ is a finite binary tree whose nodes are labelled by truncated vectors. We say $\mathcal{F}$ *derives a truncated vector* v iff $\mathcal{F}(\varepsilon) = v$. We naturally extend the notions of *compactness*, *covering witness*, and *coverability* to truncated derivations w.r.t. $\leq$.

Lemma 1. *Let $\mathcal{B} = \langle k, A, R_1, R_2 \rangle$ be a BVAS and $i \in \mathbb{N}$. For any $h \in \mathbb{N}^+$, there are finitely many truncated derivations of a BVAS of height h.*

Given a BVAS $\mathcal{B}$, we define a total ordering $\sqsubseteq$ on truncated derivations according to their heights as follows. Since for each $h \in \mathbb{N}^+$ there are only finitely many, say k_h, truncated derivations of height h, we can enumerate them without repetition, arbitrarily as $\mathcal{F}_{h1}, \ldots, \mathcal{F}_{hk_h}$. We define $\mathcal{F}_{mi} \sqsubseteq \mathcal{F}_{nj}$ iff $m < n$, or $m = n$ and $i \leq j$.

The Forest $\mathsf{Under}(\mathcal{B}, C_i)$. Given a BVAS $\mathcal{B} = \langle k, A, R_1, R_2 \rangle$ and $i \in \mathbb{N}$, we construct a forest $\mathsf{Under}(\mathcal{B}, C_i)$ whose nodes are compact truncated derivations by the following rules:

U1: For each axiom $v \in A$, the truncated derivation $\overline{v}$ is a root.
U2: Let $\mathcal{F}_1$ be a compact truncated derivation in the forest. Let $\mathcal{F}$ be a truncated derivation obtained by applying a unary rule $\delta_1 \in R_1$ to $\mathcal{F}_1$ (as in rule T2). If $\mathsf{compact}(\mathcal{F})$ has not been added to the forest then add $\mathsf{compact}(\mathcal{F})$ as a child of $\mathcal{F}_1$ in the forest.
U3: Suppose compact truncated derivations $\mathcal{F}_1, \mathcal{F}_2$ are in the forest. Let $\mathcal{F}$ be a truncated derivation obtained by applying a binary rule $\delta_2 \in R_2$ to $\mathcal{F}_1$ and $\mathcal{F}_2$ (as in rule T3). If $\mathsf{compact}(\mathcal{F})$ has not been added to the forest then we add $\mathsf{compact}(\mathcal{F})$ to the forest as a child of $\mathcal{F}'$ where $\mathcal{F}'$ is the greater one between $\mathcal{F}_1$ and $\mathcal{F}_2$ w.r.t. the total order $\sqsubseteq$.

The following lemma shows that the construction of $\mathsf{Under}(\mathcal{B}, C_i)$ eventually terminates, and that it can be used to prove coverability.

Theorem 1 (Underapproximation). *Let $\mathcal{B}$ be a BVAS.*

1. *For any $i \in \mathbb{N}$, the forest $\mathsf{Under}(\mathcal{B}, C_i)$ is finite.*
2. *Given an $i \in \mathbb{N}$, for any truncated derivation $\mathcal{F}$, there is a derivation $\mathcal{D}$ in $\mathcal{B}$ such that $\mathcal{F}(\varepsilon) \leq \mathcal{D}(\varepsilon)$.*
3. *For any vector $v \in \mathbb{N}^k$, we have $v \in \mathit{Reach}(\mathcal{B})\downarrow$ iff there exists $i \in \mathbb{N}$ such that there is a truncated derivation $\mathcal{F}$ in $\mathsf{Under}(\mathcal{B}, C_i)$ with $v \leq \mathcal{F}(\varepsilon)$.*

Proof. Part (1). Fix i. It is easy to see that there are finitely many trees in the forest and each tree is finitely branching, since there are at most finitely many trees of a given height. If the forest is not finite, then by König's lemma, there is an infinite simple path of compact truncated derivations $\mathcal{F}_1, \mathcal{F}_2, \ldots$ in the forest such that for every $i \geq 1$, $\mathcal{F}_i$ is a sub-compact truncated derivation of $\mathcal{F}_{i+1}$. This induces an infinite sequence of truncated vectors $\mathcal{F}_1(\varepsilon), \mathcal{F}_2(\varepsilon) \ldots$ such that for every $i \neq j$, $\mathcal{F}_i(\varepsilon) \neq \mathcal{F}_j(\varepsilon)$. However, since for all $\mathcal{F}$ in the forest, $\mathcal{F}(\varepsilon) \in C_i$ and C_i is finite, such infinite sequence of truncated vectors does not exist.

Part (2). By induction on the height of $\mathcal{F}$.

Part (3). $\Rightarrow$: Since $Reach(\mathcal{B}) \cap v\uparrow \neq \emptyset$, there is a derivation $\mathcal{D}$ in $\mathcal{B}$ such that $v \leq \mathcal{D}(\varepsilon)$. Let S be the union of the set of axioms A and the set of all vectors in $\mathsf{compact}(\mathcal{D})$. Because both sets are finite, let i be $\max(S)$. Then $\mathsf{compact}(\mathcal{D})$ is in $\mathsf{Under}(\mathcal{B}, C_i)$ and $v \leq \mathcal{D}(\varepsilon) = \mathsf{compact}(\mathcal{D})(\varepsilon)$.

$\Leftarrow$: By Part (2), there is a derivation $\mathcal{D}$ in $\mathcal{B}$ such that $\mathcal{F}(\varepsilon) \leq \mathcal{D}(\varepsilon)$. Since $\mathcal{D}(\varepsilon) \in Reach(\mathcal{B})$ and $v \leq \mathcal{F}(\varepsilon)$, $v \in Reach(\mathcal{B})\downarrow$. ∎

3.2 Overapproximation

To define over-approximation of derivations, we introduce *extended derivations* which consider vectors over $\mathbb{N} \cup \{\infty\}$. We then present an algorithm that builds a forest overapproximating the downward closure of reachable vectors of a given BVAS and prove termination and correctness.

Let $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$ be the extension of the natural numbers with infinity. An *extended vector* is an element of $\mathbb{N}_\infty^k$. For extended vectors $u, u' \in \mathbb{N}_\infty^k$, we write $u \leq_e u'$ iff for all $i \in [1, k]$, we have $u[i] \leq u'[i]$ or $u'[i] = \infty$. We write $u <_e u'$ iff $u \leq_e u'$ and $u' \not\leq_e u$. We always use words starting with the letter u to denote an extended vector (e.g. u, u', u_1 etc.) and words starting with the letter v to denote a vector in $\mathbb{Z}^k$ (e.g. v, v', v_1 etc.). Extended vectors describe sets of vectors: we define $\gamma : \mathbb{N}_\infty^k \to 2^{\mathbb{N}^k}$ as $\gamma(u) = \{v \in \mathbb{N}^k \mid v \leq_e u\}$, and naturally extend γ to sets of extended vectors.

Proposition 1. *[7] (1) Given an extended vector $u \in \mathbb{N}_\infty^k$ and a finite set of extended vectors $S \subseteq \mathbb{N}_\infty^k$, $\gamma(u) \subseteq \gamma(S)$ iff there is $u' \in S$ such that $u \leq_e u'$. (2) Given two finite and minimal sets $S_1, S_2 \subseteq \mathbb{N}_\infty^k$, $S_1 = S_2$ if and only if $\gamma(S_1) = \gamma(S_2)$.*

Given a BVAS $\mathcal{B} = \langle k, A, R_1, R_2 \rangle$, there exists a finite and minimal subset $\mathsf{CS}(\mathcal{B}) \subseteq \mathbb{N}_\infty^k$ such that $\gamma(\mathsf{CS}(\mathcal{B})) = Reach(\mathcal{B})\downarrow$. We shall call $\mathsf{CS}(\mathcal{B})$ the *finite representation* of $Reach(\mathcal{B})\downarrow$.

Extended Derivations. Given a BVAS $\mathcal{B} = \langle k, A, R_1, R_2 \rangle$ and an $i \in \mathbb{N}$, let $C_i = \{0, \ldots, i\}^k \cup A$ and $L_i = \{0, \ldots, i, \infty\}^k \setminus \{0, \ldots, i\}^k$. Given two sets $S_1 \subseteq \mathbb{N}^k$ and $S_2 \subseteq \mathbb{N}_\infty^k$, we say that S_2 is an *overapproximation* of S_1 iff $S_1 \subseteq \gamma(S_2)$. Moreover, we say that S_2 is *the most precise overapproximation* of S_1 in $L_i \cup C_i$ iff there is no finite and minimal subset $S \subseteq L_i \cup C_i$ such that $S_1 \subseteq \gamma(S) \subset \gamma(S_2)$. In the following, in case S_2 is a singleton set $\{u\}$, we write that u is (the most precise) overapproximation of S_1 for simplicity.

Given an extended vector $u \in \mathbb{N}_\infty^k$ and an $i \in \mathbb{N}$, We write $\mathsf{over}(u, i)$ for the extended vector such that for all $j \in [1, k]$, $\mathsf{over}(u, i)[j] = u[j]$ if $u[j] \leq i$, $\mathsf{over}(u, i)[j] = \infty$ otherwise. Note that $\mathsf{over}(u, i)$ is an overapproximation of $\gamma(u)$, and interestingly, is the most precise overapproximation of $\gamma(u)$ in $L_i \cup C_i$ [7].

We can naturally extend the addition of vectors to the addition of extended vectors by assuming that $\infty + \infty = \infty$ and $\infty + c = \infty$ for all $c \in \mathbb{Z}$.

Given a BVAS $\mathcal{B} = (k, A, R_1, R_2)$ and $i \in \mathbb{N}$, an *extended derivation* $\mathcal{E}$ is defined inductively as follows.

E1: If $v \in A$, then $\overline{v}$ is an extended derivation.

E2: If $\mathcal{E}_1$ is an extended derivation with a derived extended vector $u_1 \in \mathbb{N}^k_\infty$, then for each unary rule $\delta_1 \in R_1$ with $\mathbf{0} \leq_e u_1 + \delta_1$,

$$\mathcal{E} : \dfrac{\begin{array}{c}\vdots\,\mathcal{E}_1\\ u_1\end{array}}{u}\,\delta_1$$

is an extended derivation, where $u = \mathsf{over}(u_1 + \delta_1, i)$.

E3: If $\mathcal{E}_1$ and $\mathcal{E}_2$ are extended derivations with derived extended vectors $u_1, u_2 \in \mathbb{N}^k_\infty$ respectively, then for each binary rule $\delta_2 \in R_2$ with $\mathbf{0} \leq_e u_1 + u_2 + \delta_2$,

$$\mathcal{E} : \dfrac{\begin{array}{cc}\vdots\,\mathcal{E}_1 & \vdots\,\mathcal{E}_2\\ u_1 & u_2\end{array}}{u}\,\delta_2$$

is an extended derivation, where $u = \mathsf{over}(u_1 + u_2 + \delta_2, i)$.

Analogously to derivations, an extended derivation $\mathcal{E}$ is a finite binary tree whose nodes are labelled by extended vectors. For an extended derivation $\mathcal{E}$ and its node n, we write $\mathcal{E}(n)$ for the extended vector labelled at n. We say $\mathcal{E}$ *derives an extended vector* u iff $\mathcal{E}(\varepsilon) = u$. We naturally extend the notions of *compactness*, *covering witness*, and *coverability* to extended derivations w.r.t. $\leq_e$. Similar to derivations, the following lemma shows that there are finitely many extended derivations of a given height.

Lemma 2. *Given a BVAS $\mathcal{B} = \langle k, A, R_1, R_2\rangle$ and $i \in \mathbb{N}$, for each $h \in \mathbb{N}^+$, there are finitely many extended derivations of height h.*

Given a BVAS $\mathcal{B}$, we define a total ordering $\sqsubseteq_e$ on extended derivations according to their heights. Since for each $h \in \mathbb{N}^+$ there are only finitely many, say k_h, extended derivations of height h, we can enumerate them without repetition, arbitrarily as $\mathcal{E}_{h1}, \ldots, \mathcal{E}_{hk_h}$. We define $\mathcal{E}_{mi} \sqsubseteq_e \mathcal{E}_{nj}$ iff $m < n$, or $m = n$ and $i \leq j$.

The Forest $\mathsf{Over}(\mathcal{B}, L_i, C_i)$. Given a BVAS $\mathcal{B} = \langle k, A, R_1, R_2\rangle$ and an $i \in \mathbb{N}$, we construct a forest $\mathsf{Over}(\mathcal{B}, L_i, C_i)$ whose nodes are compact extended derivations by following the rules below.

O1: For each axiom $v \in A$, the extended derivation $\overline{v}$ is a root.

O2: If a compact extended derivation $\mathcal{E}_1$ is already in the forest and $\mathsf{compact}(\mathcal{E})$ has not been added in the forest where $\mathcal{E}$ is computed by applying a unary rule to $\mathcal{E}_1$ as in Rule E2, then add $\mathsf{compact}(\mathcal{E})$ as a child of $\mathcal{E}_1$ in the forest.

O3: If compact extended derivations $\mathcal{E}_1, \mathcal{E}_2$ are already in the forest and $\mathsf{compact}(\mathcal{E})$ has not been added in the forest where $\mathcal{E}$ is computed by applying a binary rule to $\mathcal{E}_1$ and $\mathcal{E}_2$ as in Rule E3, then we add $\mathsf{compact}(\mathcal{E})$ to the forest as a child of $\mathcal{E}'$ where $\mathcal{E}'$ is the greater one between $\mathcal{E}_1$ and $\mathcal{E}_2$ w.r.t. the total order $\sqsubseteq_e$.

Algorithm 1. EEC Algorithm to decide the coverability problem of BVAS.

Input: A BVAS $\mathcal{B} = \langle k, A, R_1, R_2 \rangle$ and a vector $t \in \mathbb{N}^k$.
Output: "Cover" if t is coverable in $\mathcal{B}$, "Uncover" otherwise.
begin
  $i \longleftarrow 0$
  while *true* **do**
    Compute $\mathsf{Under}(\mathcal{B}, C_i)$ // Expand
    Compute $\mathsf{Over}(\mathcal{B}, L_i, C_i)$ // Enlarge
    // Check
    if $\exists \mathcal{F} \in \mathsf{Under}(\mathcal{B}, C_i).\ t \leq \mathcal{F}(\varepsilon)$ **then**
      return *"Cover"*
    else if $\forall \mathcal{E} \in \mathsf{Over}(\mathcal{B}, L_i, C_i).\ t \not\leq_e \mathcal{E}(\varepsilon)$ **then**
      return *"Uncover"*
    $i \longleftarrow i + 1$

Theorem 2 (Overapproximation). *Let $\mathcal{B}$ be a BVAS.*

1. *For each $i \in \mathbb{N}$, the forest $\mathsf{Over}(\mathcal{B}, L_i, C_i)$ is finite.*
2. *Given $i \in \mathbb{N}$, for any derivation $\mathcal{D}$, there is a compact extended derivation $\mathcal{E}$ in $\mathsf{Over}(\mathcal{B}, L_i, C_i)$ with $\mathcal{D}(\varepsilon) \leq_e \mathcal{E}(\varepsilon)$.*
3. *For $v \in \mathbb{N}^k$, $Reach(\mathcal{B}) \cap v{\uparrow} = \emptyset$ iff there exists an $i \in \mathbb{N}$ such that for any compact extended derivation $\mathcal{E}$ in $\mathsf{Over}(\mathcal{B}, L_i, C_i)$, we have $\gamma(\mathcal{E}(\varepsilon)) \cap v{\uparrow} = \emptyset$.*

Proof. The proof of Part (1) is similar to the proof of Theorem 1(1), because $L_i \cup C_i$ is finite.

The proof of Part (2) is by induction on the height of $\mathcal{D}$.

Part (3). $\Leftarrow$: Suppose $Reach(\mathcal{B}) \cap v{\uparrow} \neq \emptyset$. Then there is a derivation $\mathcal{D}$ in $\mathcal{B}$ such that $\mathcal{D}(\varepsilon) \in v{\uparrow}$. Using Part (2), we can find $\mathcal{E}$ in $\mathsf{Over}(\mathcal{B}, L_i, C_i)$ such that $\mathcal{D}(\varepsilon) \leq_e \mathcal{E}(\varepsilon)$. For $\mathcal{E}$, we have $\mathcal{D}(\varepsilon) \in \gamma(\mathcal{E}(\varepsilon))$ and thus $\gamma(\mathcal{E}(\varepsilon)) \cap v{\uparrow} \neq \emptyset$.

$\Rightarrow$: Since $Reach(\mathcal{B}) \cap v{\uparrow} = \emptyset$ iff $Reach(\mathcal{B}){\downarrow} \cap v{\uparrow} = \emptyset$, $\gamma(\mathsf{CS}(\mathcal{B})) \cap v{\uparrow} = \emptyset$. Take $i \in \mathbb{N}$ such that $\mathsf{CS}(\mathcal{B}) \subseteq L_i \cup C_i$. For every extended derivation $\mathcal{E}$ in $\mathcal{B}$, we have $\gamma(\mathcal{E}(\varepsilon)) \subseteq \gamma(\mathsf{CS}(\mathcal{B}))$. This can be proved by induction on the height of $\mathcal{E}$.

For every compact extended derivation $\mathcal{E}$ in $\mathsf{Over}(\mathcal{B}, L_i, C_i)$, we therefore have that $\gamma(\mathcal{E}(\varepsilon)) \subseteq \gamma(\mathsf{CS}(\mathcal{B}))$. Hence $\gamma(\mathcal{E}(\varepsilon)) \cap v{\uparrow} = \emptyset$. ∎

3.3 EEC Algorithm

Algorithm 1 shows the schematic of the EEC algorithm. It takes as input a BVAS $\mathcal{B}$ and a target vector t. It uses an abstraction parameter i, initially 0, and defines the family of abstractions C_i and L_i. It iteratively computes the under-approximation Under and over-approximation Over w.r.t. i. If the under-approximation covers t, it returns "Cover"; if the over-approximation shows t cannot be covered, it returns "Uncover." Otherwise, it increments i and loops again. From Theorems 1 and 2, we conclude that this algorithm eventually terminates with the correct result.

We briefly remark on two optimizations. First, instead of explicitly keeping forests of derivations in Over and Under, we can only maintain the vectors that label the roots of the derivations. The structure of the forest was required to prove termination in [16], but can be reconstructed using only the vectors and the timestamps at which the vectors were added. Second, in Under (resp. Over), we can only keep maximal vectors (resp. extended vectors): if two vectors $v_1 \leq v_2$ (resp. extended vectors $u_1 \leq_e u_2$), we can omit v_1 (resp. u_1) and only keep v_2 (resp. u_2). Indeed, if $t \leq v_1$ in Under, we also have $t \leq v_2$, and so the cover check succeeds in the EEC algorithm. Further, if $t \not\leq_e u_2$ in Over, we have $t \not\leq_e u_1$, and so the uncover check succeeds as well. We thank Sylvain Schmitz for these observations.

4 Complexity Analysis

We now give an upper bound on the number of iterations of the EEC algorithm. Given a BVAS $\mathcal{B} = \langle k, A, R_1, R_2 \rangle$ and a derivation $\mathcal{D}$, for each internal node n, we write $\delta(n) \in \mathbb{Z}^k$ for the rule $\delta \in R_1 \cup R_2$ that is applied to derive $\mathcal{D}(n)$. We extend this notation to truncated and extended derivations as well. Given a derivation $\mathcal{D}$ and an $i \in \mathbb{N}^k$, we define a truncated derivation $\mathsf{under}(\mathcal{D}, i)$ inductively as follows:

1. If n is a leaf, then $\mathsf{under}(\mathcal{D}, i)(n) = \mathcal{D}(n)$.
2. If n has a child n' and $\mathcal{D}(n) = \mathcal{D}(n') + \delta(n)$, then $\mathsf{under}(\mathcal{D}, i)(n) = \mathsf{under}(\mathsf{under}(\mathcal{D}, i)(n') + \delta(n), i)$.
3. If n has two children n', n'' and $\mathcal{D}(n) = \mathcal{D}(n') + \mathcal{D}(n'') + \delta(n)$, then $\mathsf{under}(\mathcal{D}, i)(n) = \mathsf{under}(\mathsf{under}(\mathcal{D}, i)(n') + \mathsf{under}(\mathcal{D}, i)(n'') + \delta(n), i)$.

We can also define an extended derivation $\mathsf{over}(\mathcal{D}, i)$ inductively by following the above rules except that we replace all $\mathsf{under}(\Box, i)$ by $\mathsf{over}(\Box, i)$.

We start with some intuition in the special case of vector addition systems. A *vector addition system* (VAS) $\mathcal{V}$ is a BVAS $\langle k, \{a\}, R, \emptyset \rangle$. For simplicity, we write a VAS as just $\langle k, a, R \rangle$. Note that a derivation $\mathcal{D}$ of a VAS $\mathcal{V}$ is degenerated to a sequence of non-negative vectors. In the following, we say the *length* of $\mathcal{D}$ instead of the height of $\mathcal{D}$ for convenience in the VAS context. For VAS, Rackoff [15] proved the coverability problem is EXPSPACE-complete by showing that if a covering witness (derivation) exists, then there must exist one whose length h is at most doubly exponential in the size of the VAS $\mathcal{V}$ and the target vector t. Further, there is a derivation of length at most h in which the maximum constant is bounded by $i := h \cdot \mathit{size}(\mathcal{V}) + \max(t)$. This is because in h steps, a vector can decrease at most $h \cdot \mathit{size}(\mathcal{V})$, so if any co-ordinate goes over i, it remains higher than $\max(t)$ after executing the path. By the same argument, if there is an extended derivation of length at most h and constant i covering t, then we can find a derivation for t.

If t is coverable, using the above argument and Theorem 1, we see that $\mathsf{Under}(\mathcal{V}, C_i)$ will contain a covering witness of t. If t is not coverable, then the above argument shows that all extended derivations of $\mathsf{Over}(\mathcal{V}, L_i, C_i)$ of length at most h will not cover t. However, there may be longer extended derivations

in $\mathsf{Over}(\mathcal{V}, L_i, C_i)$. For these, we can show that $\mathsf{Over}(\mathcal{V}, L_i, C_i)$ also contains a contraction of that extended derivation of length at most h. In both cases, EEC terminates in i iterations, which is doubly exponential in the size of the input.

We now show the bound for BVAS. The following lemma is the key observation in the optimal algorithm of [5].

Lemma 3. *[5] Given a BVAS $\mathcal{B} = \langle k, A, R_1, R_2\rangle$ and a vector $t \in \mathbb{N}^k$, if t is coverable in $\mathcal{B}$, then there is a covering witness (derivation) $\mathcal{D}$ whose height is at most* $(\max((R_1 \cup R_2)^-) + \max(t) + 2)^{(3k)!}$.

Moreover, the following lemma shows that the maximum constant appearing in a height-bounded derivation can remain polynomial in the height.

Lemma 4. *Given a BVAS $\mathcal{B} = \langle k, A, R_1, R_2\rangle$, a vector $t \in \mathbb{N}^k$ and a derivation $\mathcal{D}$ whose height is at most h, for any bound $i \geq h \cdot \max((R_1 \cup R_2)^-) + \max(t)$, $\mathcal{D}$ is a covering witness of t iff $\mathsf{under}(\mathcal{D}, i)$ is a covering witness of t.*

Proof. Fix an i such that $i \geq h \cdot \max((R_1 \cup R_2)^-) + \max(t)$.

$\Leftarrow$: It holds by Theorem 1.

$\Rightarrow$: Given a derivation $\mathcal{D}$, we say that an index j is *marked* iff during the construction of $\mathsf{under}(\mathcal{D}, i)$, there is a vector v, which is computed after applying a rule and before comparing to i, such that $v[j] > i$.

Given a derivation $\mathcal{D}$, during the construction of $\mathsf{under}(\mathcal{D}, i)$, for each index $j \in [1, k]$, we check the following: If j is marked, then there is a node n such that $\mathsf{under}(\mathcal{D}, i)(n)[j] = i$. Since $height(\mathsf{under}(\mathcal{D}, i)) = height(\mathcal{D}) \leq h$, we know that the length of the path from n to the root ε is at most h. Hence $\mathsf{under}(\mathcal{D}, i)(\varepsilon)[j] \geq \mathsf{under}(\mathcal{D}, i)(n)[j] - h \cdot \max((R_1 \cup R_2)^-) = i - h \cdot \max((R_1 \cup R_2)^-) \geq \max(t) \geq t[j]$. On the other hand, if j is not marked, we have that for all node n, $\mathsf{under}(\mathcal{D}, i)(n)[j] = \mathcal{D}(n)[j]$. Hence $\mathsf{under}(\mathcal{D}, i)(\varepsilon)[j] = \mathcal{D}(\varepsilon)[j] \geq t[j]$. Hence $\mathsf{under}(\mathcal{D}, i)$ is a covering witness of t. ∎

We now prove the case where the target vector t is coverable. We show that $\mathsf{Under}(\mathcal{B}, C_i)$ contains a truncated derivation covering t, where i is bounded by a doubly exponential function of the input.

Lemma 5. *Given a BVAS $\mathcal{B} = \langle k, A, R_1, R_2\rangle$ and a vector $t \in \mathbb{N}^k$, if t is coverable in $\mathcal{B}$, then there exists $\mathcal{F} \in \mathsf{Under}(\mathcal{B}, C_i)$ such that $t \leq \mathcal{F}(\varepsilon)$ for some $i = 2^{2^{O(n \log n)}}$, where $n = size(\mathcal{B}) + size(t)$.*

Proof. Let h be the bound from Lemma 3. Clearly, $h = 2^{2^{O(nlogn)}}$. Pick $i = h^2$. By Lemma 3, there is a derivation $\mathcal{D}$ that covers t and whose height is at most h. Since $i = h^2 \geq h \cdot \max((R_1 \cup R_2)^-) + \max(t)$, by Lemma 4, there is a truncated derivation $\mathsf{under}(\mathcal{D}, i)$ that covers t. Moreover, $\mathsf{compact}(\mathsf{under}(\mathcal{D}, i))$ is in $\mathsf{Under}(\mathcal{B}, C_i)$. ∎

Assume now that the target vector $t \in \mathbb{N}^k$ is not coverable. Lemma 6, from [5], connects derivations of "small" height to extended derivations for high enough constants. Lemma 7 shows that extended derivations of "large" height can be contracted. The proof of this lemma mimicks the proof for (ordinary) derivations.

Lemma 6. *[5] Given a BVAS $\langle k, A, R_1, R_2\rangle$, a vector $t \in \mathbb{N}^k$, and a derivation $\mathcal{D}$ whose height is at most h, for any bound $i \geq h \cdot \max((R_1 \cup R_2)^-) + \max(t)$, $\mathcal{D}$ is a covering witness of t iff $\mathsf{over}(\mathcal{D}, i)$ is a covering witness of t.*

Lemma 7. *Let $\mathcal{B} = \langle k, A, R_1, R_2\rangle$ be a BVAS and $i \in \mathbb{N}$. If there is an extended derivation $\mathcal{E}$ that covers $t \in \mathbb{N}^k$, then there is a contraction of $\mathcal{E}$ whose height is at most $(\max((R_1 \cup R_2)^-) + \max(t) + 2)^{(3k)!}$.*

Finally, we prove that if t is not coverable, then $\mathsf{Over}(\mathcal{B}, L_i, C_i)$ does not find an extended derivation covering t, for i as above.

Lemma 8. *Given a BVAS $\mathcal{B} = \langle k, A, R_1, R_2\rangle$ and $t \in \mathbb{N}^k$, there is an $i = 2^{2^{O(n \log n)}}$, where $n = size(\mathcal{B}) + size(t)$, such that if t is not coverable in $\mathcal{B}$, then for all extended derivations $\mathcal{E} \in \mathsf{Over}(\mathcal{B}, L_i, C_i)$, we have $\mathcal{E}$ does not cover t.*

Proof. Suppose not. Then there is an $\mathcal{E} \in \mathsf{Over}(\mathcal{B}, L_i, C_i)$ so that $\mathcal{E}$ covers t. Let h be the bound from Lemma 7, and let $i = h^2$. We consider two cases: (1) The height of $\mathcal{E}$ is at most h. Then since $i = h^2 \geq h \cdot \max((R_1 \cup R_2)^-) + \max(t)$, by Lemma 6, t is coverable in $\mathcal{B}$. Contradiction. (2) The height of $\mathcal{E}$ is greater than h. By Lemma 7, there is a contraction of $\mathcal{E}$ that covers t and whose height is at most h. Following the arguments in case (1), we again get a contradiction. ∎

Our main theorem follows from Lemmas 5 and 8.

Theorem 3. *Given a BVAS $\mathcal{B} = \langle k, A, R_1, R_2\rangle$ and a vector $t \in \mathbb{N}^k$, the EEC algorithm terminates in $2^{2^{O(n \log n)}}$ iterations, where $n = size(\mathcal{B}) + size(t)$.*

The bound on the number of iterations also provides a bound on the overall asymptotic complexity of the algorithm. For BVAS, each iteration of the EEC algorithm performs two instances of AND-OR reachability to perform the cover and the uncover checks. Moreover, the size of the graph is at most doubly exponential in the size of the BVAS, since the finite component of each vector is bounded by a doubly exponential function of the input. Since AND-OR reachability can be performed in time linear in the size of the graph, this gives a 2EXPTIME algorithm. For VAS, each iteration of the EEC algorithm performs two instances of reachability to perform the checks. Thus, if reachability is implemented in a space optimal (NLOGSPACE) way, we get an EXPSPACE upper bound. (In practice, reachability is implemented using a linear time algorithm, which leads to a 2EXPTIME upper bound.)

5 Implementation and Evaluation

We have implemented the EEC algorithm for BVAS and used our implementation to model check safety properties of single-wait recursively parallel programs [1]. Our programs are written in the syntax of [1], and we assume all program variables range over finite domains. In the following, we briefly recall recursively parallel programs with a *wait* construct. We then describe the performance of our tool on a web server example.

Recursively Parallel Programs. Single-wait recursively parallel programs (see [1] for details) are a concurrent programming model in which computations are hierarchically organized into isolated parallelly executing tasks. Each task executes sequentially, and maintains *regions* of *handles* to other tasks. A task t can *post* subtasks and store their handles in one of its regions. A subtask posted by t executes in parallel with the task t. A task can create unboundedly many subtasks in a single region and each subtask may also recursively create additional parallel tasks, storing their handles in its own regions. At some later point, when the task t requires the results computed by its subtasks, t has to wait until a posted subtask completes. A wait is implemented with an "ewait" construct: when task t executes $ewait(r)$ for a region r, its execution is suspended until *some* task whose handle is stored in r completes execution. The return value of the completed subtask is combined with the current state of t via a programmer-supplied *return-value handler*, and t continues executing from this point with the updated state.

The *state reachability problem* for a recursively parallel program P, is to determine, given an initial valuation l_0 of variables of the program P and a valuation l, if there is an execution of P such that the valuation l is reachable.

Single-wait programs capture many constructs in modern structural parallel programming languages, such as pruning in Orc [12], futures in Multilisp [8], task-parallel libraries [3,14], asynchronous programs [9,6], etc. The state reachability problem for single-wait programs is equivalent to the coverability problem for BVAS [1].[1]

Web Server Case Study. We implemented the EEC algorithm for BVAS and a reduction from recursively parallel programs with ewait to BVAS. We used our implementation to verify safety properties of a model of a web server taken from [4]. Our model executes asynchronous calls to serve requests, and waits on multiple concurrent requests to implement DNS lookup.

We model a server main that helps clients upload files. Given a provider's domain name and files, the server main simulates arbitrarily many client requests by posting unboundedly many dns requests. The server waits on the first DNS server to return. The DNS server returns the provider's real IP address, which is stored in a variable x.

Each task dns waits for DNS lookup requests, and returns either an ip if the lookup succeeds, or not_found if the provided domain name is not valid. If the domain name is valid, dns then either returns ip if it already has an IP address for the requested domain name by looking up its local database, or posts a task server_i to ask other servers to resolve an IP address. Before asking a remote server server_i, dns first allocates a buffer buf that is used for the communication between itself and server_i, in particular, used for storing an ip returned from server_i. When dns receives an ip stored in the buffer buf from some remote server, it returns the contents of buf. dns may ask multiple remote servers at the same time, and accept

[1] The authors of [1] state that the 2EXPTIME algorithm of Demri et al. will be hard to implement. They propose an alternate algorithm, without any upper bound on the running time. However, that algorithm has not been implemented either.

Table 1. Results of the DNS example

#server	#dimension	#axiom	#urule	#brule	#iter	result	time
2	19	20	9153	6950	1	Uncover	31.25s
3	22	23	14832	11664	1	Uncover	79.16s
4	25	26	22640	18326	1	Uncover	151.46s
5	28	29	33070	27392	1	Uncover	279.56s
6	31	32	46638	39366	1	Uncover	463.71s
6 (buggy)	31	32	40077	32805	1	Cover	63.58s

the very first *ip* returned from some server. We model this scenario precisely by posting all server tasks into a single region and waiting for the first one to return its result (using ewait). A task server_i can either return a *timeout* to model the cases where server_i encounters a problem, or an *ip* when server_i successfully completes a lookup. Since server_i can produce *timeout*, the task dns contains error handling mechanism: if it receives a *timeout*, it de-allocates the buffer buf.

The first property we check is that the value stored in x when task dns returns always equals either an *ip* or a not_found.

If the server main gets the provider's IP address successfully, then it uploads the file on the client's behalf by allocating and using a buffer for the transmission. Since errors may occur during the transmission, the server provides a block of code for error handling: whenever an error happens, the server does some cleaning work such as de-allocating the buffer. Once the transmission is over, the server main receives an acknowledgement in the buffer and returns this buffer to the client, informing the client that the file has been successfully uploaded. The second property we check is if the buffer is always de-allocated properly after the error handling block completes.

Since any task can create subtasks and moreover the task main creates unboundedly many subtasks, by the classification from [1], this example falls into the general case of single-wait programs which are equivalent to BVAS.

Results. We have run our tool to verify the two properties above. All experiments were performed on a 2 core Intel Xeon X5650 CPU machine with 64GB memory and 64bit Linux (Debian/Lenny). Table 1 lists the analysis results of both cases. We report: (1) the size of the generated BVAS (the dimension, the number of axioms, unary rules, and binary rules), (2) the number of iterations i for EEC, and (3) the answer, "Cover" or "Uncover," and the execution time.

We modeled a bug found in [4] where the task main can receive a value in the variable x that is neither *ip* nor not_found from dns. In the buggy version, in the error handling code of the task dns, the programmer forgot to return to its caller on an error after de-allocating a buffer. Therefore, after the de-allocation of the buffer, the dns lookup continued to execute and returned the value of the buffer (in the normal case, the buffer contains an *ip*). After we added an immediate return to dns, our tool proved the model is correct.

To explore the scalability of our implementation, we increased the number of remote servers (as shown in the first column of the table) which a task dns

posts to a single region in parallel. The "Uncover" instances in Table 1 use the corrected version of dns and increase the number of servers from 2 to 6. As the size of the BVASs becomes larger, the EEC algorithm takes more time to verify instances. However, the largest example, with 31 dimensions, still finishes within a few minutes. We present the run time of the unsafe case when the number of servers is six. For fewer servers, the bug is found quicker. Additionally, for these examples, one iteration of the EEC algorithm (i.e., the counter value is in $\{0, 1, \infty\}$) is sufficient to prove or disprove the property.

References

1. Bouajjani, A., Emmi, M.: Analysis of recursively parallel programs. In: POPL, pp. 203–214 (2012)
2. Bozzelli, L., Ganty, P.: Complexity analysis of the backward coverability algorithm for VASS. In: Delzanno, G., Potapov, I. (eds.) RP 2011. LNCS, vol. 6945, pp. 96–109. Springer, Heidelberg (2011)
3. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not. 40(10), 519–538 (2005)
4. Cunningham, R.: Eel: Tools for debugging, visualization, and verification of event-driven software. Master's thesis, UCLA (2005)
5. Demri, S., Jurdzinski, M., Lachish, O., Lazic, R.: The covering and boundedness problems for branching vector addition systems. J. Comput. Syst. Sci. 79(1), 23–38 (2013)
6. Ganty, P., Majumdar, R.: Algorithmic verification of asynchronous programs. In: ACM Transactions on Programming Languages and Systems (2012)
7. Geeraerts, G., Raskin, J.-F., Begin, L.V.: Expand, enlarge and check: New algorithms for the coverability problem of wsts. J. Comput. Syst. Sci. 72(1), 180–203 (2006)
8. Halstead, R.H.: Multilisp: a language for concurrent symbolic computation. ACM Transactions on Programming Languages and Systems 7, 501–538 (1985)
9. Jhala, R., Majumdar, R.: Interprocedural analysis of asynchronous programs. In: POPL 2007, pp. 339–350. ACM Press (2007)
10. Kaiser, A., Kroening, D., Wahl, T.: Efficient coverability analysis by proof minimization. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 500–515. Springer, Heidelberg (2012)
11. Karp, R., Miller, R.: Parallel program schemata. Journal of Comput. Syst. Sci. 3(2), 147–195 (1969)
12. Kitchin, D., Quark, A., Cook, W., Misra, J.: The Orc programming language. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS 2009. LNCS, vol. 5522, pp. 1–25. Springer, Heidelberg (2009)
13. Kloos, J., Majumdar, R., Niksic, F., Piskac, R.: Incremental, inductive coverability. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 158–173. Springer, Heidelberg (2013)
14. Leijen, D., Schulte, W., Burckhardt, S.: The design of a task parallel library. In: OOPSLA 2009, pp. 227–242. ACM (2009)
15. Rackoff, C.: The covering and boundedness problems for vector addition systems. Theoretical Computer Science 6(2), 223–231 (1978)
16. Verma, K., Goubault-Larrecq, J.: Karp-Miller trees for a branching extension of VASS. Discrete Mathematics & Theoretical Computer Science 7(1), 217–230 (2005)
17. Verma, K.N., Goubault-Larrecq, J.: Alternating two-way AC-tree automata. Inf. Comput. 205(6), 817–869 (2007)

Symbolic Bisimulation for a Higher-Order Distributed Language with Passivation*
(Extended Abstract)

Vasileios Koutavas and Matthew Hennessy

Trinity College Dublin
{Vasileios.Koutavas,Matthew.Hennessy}@scss.tcd.ie

Abstract. We study the behavioural theory of a higher-order distributed calculus with private names and locations that can be passivated. For this language, we present a novel Labelled Transition System where higher-order inputs are symbolic agents that can perform a limited number of transitions, capturing the nature of passivation. Standard first-order weak bisimulation over this LTS coincides with contextual equivalence, and provides the first useful proof technique without a universal quantification over contexts for an intricate distributed language.

1 Introduction

Higher-order concurrency naturally arises from the combination of functional and concurrent programming. In many concurrency scenarios, processes have the ability to exchange values over communication channels; in languages with functional characteristics, besides constants of base type, these values include code in the form of function closures of higher types.

The behaviour of processes in simple higher-order concurrency has been studied in the setting of Higher-Order π-calculus (HOπ) [11] and CHOCS [1]. The former work showed that higher-order systems can be translated and studied in first-order π-calculus [12]. The translation is based on the notion of a *trigger*, a simple value representing a function which, when run within a process, triggers the execution of the function in another part of the system. This translation gave rise to *normal bisimulation*, a first-order bisimulation method in which the observer need only examine a process using finite trigger values, enabling simple proofs of equivalence. This proof method is both sound and complete with respect to a natural contextual equivalence, called *barbed congruence* [12, 4, 6].

For *distributed systems*, however the approach of the trigger translation is generally not applicable [15, 7]. The intuition is that in many distributed scenarios, the observable runtime behaviour of a higher-order value depends on the location in which it is run. One of the simplest extensions to HOπ where this *location-dependent behaviour* becomes apparent is when *passivation* is added to the language, as in HOπP [7] which uses transparent locations whose contents

* This research was supported by SFI project SFI 06 IN.1 1898.

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 167–181, 2013.

can be passivated and restarted in a different context. This simple construct is sufficient to demonstrate the intricacies arising when communication is location dependent; for example, processes can become temporarily isolated from the external observer, or indeed from other components of the system, essentially encoding *communication barriers*. The resulting behavioural complexity is emphasised by the results for HOπP, which show that extensions of triggers to include arbitrary finite values do not capture contextual equivalence [7, Sec. 6].

The purpose of this paper is to demonstrate that contextual equivalence for distributed systems exhibiting location-dependent behaviour can be captured by a first-order bisimulation semantics in which triggers are replaced by simple *symbolic agents*. We give the first sound and complete first-order bisimulation technique for equivalence in a higher-order distributed language with passivation and private names, which avoids universal quantification over contexts.

The starting point is the *labelled transition system* (LTS) semantics in previous work for HOπ [6], encoding Sangiorgi's triggers as symbolic constants. These constants represent the actual higher-order values transmitted between the observer and the system under observation. This limits the size and complexity of the resulting LTS as these constants can only be subsequently used by the observer to run code produced by the system, and by the system to signal the execution of observer-generated symbolic code. However, for location-dependent behaviour the repertoire of symbolic constants has to be enlarged to what we call *symbolic agents*, a small collection of probes designed to facilitate two kinds of observations capturing the nature of passivation and more generally location-dependent behaviour. The first is to discover if locations in systems where agents are running can communicate with other locations and the observer. The other is to examine the system when system-emitted code runs at agent locations.

The language we consider is a minor variation of HOπP in which, because of lazy scope extrusion of π-calculus names, contextual equivalence can distinguish between systems solely on the basis of their free names [7, Sec. 2.4]. This infelicity is avoided in a variant called HOπPn [10] in which π-calculus restriction is replaced by name allocation; however, this sacrifices expressiveness since basic programming constructs such as recursion and internal choice are not programmable (see Thm. 3.4). In this paper we opt for a passivation language HOPass which, like HOπPn, avoids the complications with free names of HOπP, but also *can* encode useful programming constructs. Our language essentially adds CCS-style local communication ports to HOπPn solely for the purpose of programmability.

Both HOπP and HOπPn have coinductive characterisations of contextual equivalence, in terms of *weak context bisimulation* [7] and *weak environmental bisimulation* [10], respectively. However, the former does not provide a viable proof technique for equivalence because of a significant universal quantification over contexts. The latter also contains a similar quantification; however, powerful *up-to techniques* [13] can certainly help with constructing witness bisimulations.

The symbolic agent LTS in this paper avoids any quantification over contexts and provides a viable proof technique relying only on standard (weak) first-order

$$
\begin{array}{rrcl}
 & a,\ldots,t & \in & \mathsf{GName} \qquad p_{\mathrm{L}} \in \mathsf{LPort} \qquad x,y,z \in \mathsf{Var} \\
 & u,v & \in & \mathsf{GName} \cup \mathsf{LPort} \qquad \widehat{u},\widehat{v} \in \mathsf{Var} \cup \mathsf{GName} \cup \mathsf{LPort} \\
\mathsf{Val}: & V & ::= & a \mid \lambda P \qquad \widehat{V} \in \mathsf{Var} \cup \mathsf{Val} \\
\mathsf{Proc}: & P,Q & ::= & \mathtt{0} \mid \widehat{u}!\widehat{V}.P \mid \widehat{u}?(x{:}T).P \mid (P \mid P) \mid \mathtt{if}\ \widehat{V}=\widehat{V}\ \mathtt{then}\ P\ \mathtt{else}\ P \\
 & & & \mid \mathtt{new}\ x.P \mid P\backslash p_{\mathrm{L}} \mid \mathtt{run}\ \widehat{V} \mid \widehat{u}[\![P]\!] \\
\mathsf{Sys}: & M,N & ::= & P \mid \nu a.M \qquad \mathsf{Type}: \quad T ::= \mathsf{Nm} \mid \mathsf{Pr}
\end{array}
$$

Fig. 1. Syntax of HOPass

bisimulation. The usefulness of first-order techniques have been demonstrated for HOπ [12, 4–6], and are equally useful for HOPass. Additionally, our proof technique reduces the size of bisimulations in proofs of equivalence by minimising the number of symbolic transitions that need to be considered. Moreover, we believe that our symbolic agent semantics can be adapted to other distributed languages with location-dependent behaviour, including HOπP.

We continue with the description of HOPass (Sect. 2) and contextual equivalence (Sect. 3). We then explain the intuitions of our LTS (Sect. 4) and detail its symbolic agent transitions (Sect. 5). The sound and complete bisimulation technique and an example equivalence are given in Sect(s). 6 and 7, respectively.

2 The Language HOPass

The abstract syntax of HOPass is shown in Fig. 1. *Generated names* (GName) are used for general communication channels between processes, and *local ports* (LPort) for programming via CCS-style locally scoped communication. *Values* (Val) are the objects transmitted over channels which can be first-order generated names of type Nm or higher-order *code thunks* of type Pr. Terms in HOPass are constructed in two levels: the inner level of *processes* (Proc) and the outer level of *systems* (Sys). A process can be one of the usual π-calculus inert ($\mathtt{0}$), output ($c!V.P$), input ($c?(x{:}T).P$), parallel ($P \mid Q$), and conditional process. Because channels can carry two types of values, we use the type annotation T at input processes and a simple dynamic type system to rule out stuck processes (see [6]).

Processes can also create at runtime a fresh generated name ($\mathtt{new}\, x.P$), and restrict local ports which are CCS channels used for programmability ($P\backslash p_{\mathrm{L}}$). We will only reason about *closed processes* with no free local ports or variables but can have free generated names. Finally, a process can execute a code thunk ($\mathtt{run}\, V$) and run a process within a location u ($u[\![P]\!]$). As we will see, P can reduce inside u and can be passivated by an input on u. A system takes the form $\nu n_1.\ldots\nu n_i.P$, consisting of a single process P with a number of bound names. We use the Barendregt convention for bound generated names. Free generated names and local ports are given by $\mathrm{fn}(-)$ and $\mathrm{flp}(-)$, respectively; we use $(- \sharp -)$ to mean "have disjoint names and ports".

The reduction semantics of systems, $M \to N$, is defined in terms of labelled transitions of processes $P \xrightarrow{\lambda} N$ (Fig. 2). A process transition is annotated with a label indicating an output ($u!V$), input ($u?V$), internal transition (τ), or fresh

Process Transitions

RPIN
$$\frac{\vdash_{\mathsf{V}} V : T}{u?(x{:}T).P \xrightarrow{u?V} P\{V/x\}}$$

RPOUT
$$\frac{\mathrm{flp}(V) = \emptyset}{c!V.P \xrightarrow{c!V} P}$$

RPCOMML
$$\frac{P \xrightarrow{u!V} P' \qquad Q \xrightarrow{u?V} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

RPNEW
$$\frac{n \sharp P}{\mathtt{new}\, x.P \xrightarrow{\mathtt{new}\, n} P\{n/x\}}$$

RPPASS
$$\frac{\mathrm{flp}(V) = \emptyset}{c[\![P]\!] \xrightarrow{c!\lambda P} \mathbf{0}}$$

RPMATCH
$$\frac{}{\mathtt{if}\, a = a\, \mathtt{then}\, P\, \mathtt{else}\, Q \xrightarrow{\tau} P}$$

RPPORT
$$\frac{P \xrightarrow{\lambda} Q \qquad p_{\mathrm{L}} \sharp \lambda}{P\backslash p_{\mathrm{L}} \xrightarrow{\lambda} Q\backslash p_{\mathrm{L}}}$$

RPOUTPORT
$$\frac{}{p_{\mathrm{L}}!V.P \xrightarrow{p_{\mathrm{L}}!V} P}$$

RPMISMATCH
$$\frac{a \neq b}{\mathtt{if}\, a = b\, \mathtt{then}\, P\, \mathtt{else}\, Q \xrightarrow{\tau} Q}$$

RPRUN
$$\frac{}{\mathtt{run}\, \lambda P \xrightarrow{\tau} P}$$

RPLOC
$$\frac{P \xrightarrow{\lambda} Q}{u[\![P]\!] \xrightarrow{\lambda} u[\![Q]\!]}$$

RPPARL
$$\frac{P \xrightarrow{\lambda} P' \quad \mathrm{new}(\lambda) \sharp Q}{P \mid Q \xrightarrow{\lambda} P' \mid Q}$$

System Reductions

RSν
$$\frac{M \to M'}{\nu n.M \to \nu n.M'}$$

RSτ
$$\frac{P \xrightarrow{\tau} P'}{P \to P'}$$

RSNEW
$$\frac{P \xrightarrow{\mathtt{new}\, n} P'}{P \to \nu n.P'}$$

Fig. 2. Reduction semantics of HOPass (omitting symmetric rules and RPPASSPORT)

name generation ($\mathtt{new}\, n$). Output (RPOUT) and passivation (RPPASS) over generated names transmit closed values; over local ports (RPOUTPORT, RPPASSPORT) they can transmit values with free local ports, enabling the encoding of useful programming idioms (see Thm. 3.4) while avoiding the extrusion of local ports. Input (RPIN) receives values of the appropriate type; here $\vdash_{\mathsf{V}} \lambda P : \mathsf{Pr}$ (for any P) and $\vdash_{\mathsf{V}} n : \mathsf{Nm}$, and $P\{V/x\}$ is capture-avoiding substitution. A new generated name is fresh because of the side-conditions in RPNEW and RPPARL; in the latter rule $\mathrm{new}(\lambda)$ denotes $\{n\}$, when $\lambda = \mathtt{new}\, n$, and $\emptyset$ otherwise. The rest are standard rules for running a code thunk (RPRUN), communication (RPCOMML), equality testing (RPMATCH, RPMISMATCH), and propagating transitions over evaluation contexts (RPPARL, RPLOC, RPPORT). System reductions simply bind freshly generated names and propagate internal process transitions. In the following we will use usual syntactic abbreviations from CCS and π-calculus.

3 Contextual Equivalence

In this paper we study *barbed congruence* [8], the contextual equivalence associated with weak bisimulation, which is reduction-closed, preserves weak barbs, and is a congruence. A *weak barb* is the ability of a system to perform an output on a free channel after a number of reductions.

Definition 3.1 (Weak Barb). *M has a weak barb b, written as $M \Downarrow_b$, when $M \to^* \nu\widetilde{n}.P$ and $P \xrightarrow{b!V} Q$, for some $\widetilde{n}$, P, Q, and V with $b \sharp \widetilde{n}$.*

We use standard, single-hole contexts derived from the language grammar, adding a process hole. We use the Barendregt convention only for bound names whose scope does not extend over the hole. We write $K[P]$ to mean the system obtained by replacing the hole of K with process P, and $K[M]$ to mean $\nu\widetilde{n}.\nu\widetilde{m}.K[P]$ when $K = \nu\widetilde{n}.K$ and $M = \nu\widetilde{m}.P$. Due to the convention for the bound names $\widetilde{m}$ we have $\widetilde{m} \sharp K, \widetilde{n}$; however, $\widetilde{n}$ may appear in P.

Definition 3.2 (Contextual Equivalence ($\cong_{\text{cxt}}$)). *The relation ($\cong_{\text{cxt}}$) on systems is the largest relation such that if $M \cong_{\text{cxt}} M'$ then*

1. *For all b, $M \Downarrow_b$ iff $M' \Downarrow_b$.*
2. *If $M \to N$ then there exists N' such that $M' \to^* N'$ and $N \cong_{\text{cxt}} N'$.*
3. *If $M' \to N'$ then there exists N such that $M \to^* N$ and $N \cong_{\text{cxt}} N'$.*
4. *For any context K_{S}, $K_{\mathsf{S}}[M] \cong_{\text{cxt}} K_{\mathsf{S}}[M']$.*

If we remove passivation from this language we obtain a language similar to HOπ [11]. It is known that for such a language ($\cong_{\text{cxt}}$) coincides with the version of contextual equivalence which only requires preservation of the relation under parallel contexts (cf. Thm. 3.2). As the following example shows, this is not the case in the presence of passivation.

Example 3.3 (Passivation). Consider the systems

$$M_{3.3} = a!(\lambda b!).\mathbf{0} \qquad\qquad M'_{3.3} = \mathtt{new}\, k.\,(a!(\lambda k!).*(k?b!))$$

These systems are indistinguishable if we consider only parallel contexts. The intuition is that both systems output a code thunk to any parallel context. The former outputs $\lambda b!$, becoming $\mathbf{0}$, and the latter outputs $\lambda k!$, leaving behind $*(k?b!)$, a replicated process defined in Thm. 3.4. In the case of $M_{3.3}$, the parallel context can essentially only run $\lambda b!$ producing a b-barb (possibly multiple times). Because k is never revealed to the context, in the case of $M'_{3.3}$ whenever the parallel context runs $\lambda k!$ it will again trigger a b-barb. Thus, no parallel context is able to induce an observable difference between $M_{3.3}$ and $M'_{3.3}$.

However, ($\cong_{\text{cxt}}$) considers contexts that run $M_{3.3}$ and $M'_{3.3}$ in a location l, enabling the passivation of process $*(k?b!)$ and distinguishing the behaviour of the two systems. The distinguishing context $K_{3.3} = l[\![\,[\cdot]\,]\!] \mid a?(x).\,l?.\,\mathtt{run}\,x$ can input the code from channel a, passivate location l, and run the received code. Thus, $K_{3.3}[M_{3.3}] \Downarrow_b$ but $K_{3.3}[M'_{3.3}] \not\Downarrow_b$ because the latter reduces to $\nu k.k!$. □

Example 3.4 (Derived programming constructs). Consider an extension of HOPass with standard operation of internal choice $(- \oplus -)$ with the non-deterministic reduction semantics: $P \oplus Q \to P$ and $P \oplus Q \to Q$. In this extended language the definition of ($\cong_{\text{cxt}}$) (Thm. 3.2) still applies. We can implement the internal choice operator correctly using local ports. Using the bisimulation technique developed in Sect. 6, one can show that

$$P \oplus Q \;\cong_{\text{cxt}}\; (p_{\text{L}}! \mid p_{\text{L}}?.P \mid p_{\text{L}}?.Q)\backslash p_{\text{L}}$$

An implementation using generated names

$$P \oplus_{\mathtt{new}} Q \stackrel{\text{def}}{=} \mathtt{new}\, x.(x! \mid x?.P \mid x?.Q)$$

would be incorrect because in general $P \oplus Q \not\cong_{\text{cxt}} P \oplus_{\mathtt{new}} Q$. To see this consider a particular instance when P, Q are $a!$, $b!$, respectively; we show that $a! \oplus_{\mathtt{new}} b! \not\cong_{\text{cxt}} a! \oplus b!$. The idea is to place the processes in a location $l[\![\ ]\!]$ which can be passivated and duplicated. Consider $R_l = l[\![a! \oplus b!]\!]$. If we run R_l in parallel with process $K_1 = l?(x).(l'[\![\mathtt{run}\, x]\!] \mid l'[\![\mathtt{run}\, x]\!])$ the code in location l will be passivated and duplicated giving us $R_{l'} \mid R_{l'}$. Moreover, process $K_2 = l'?.a?.b?.c!$ blocks when run in parallel with $R_{l'} \mid R_{l'}$. By putting the above processes together we have $R_l \mid K_1 \mid K_2 \not\Downarrow_c$.

However, when $R'_l = l[\![a! \oplus_{\mathtt{new}} b!]\!]$ runs, a fresh name k is generated and we obtain $\nu k.l[\![k! \mid k?.a! \mid k?.b!]\!]$. Thus $R'_l \mid K_1 \mid K_2 \Downarrow_c$ because it can evolve to

$$\begin{array}{lllll}
 & \nu k.\ l'[\![k! \mid k?.a! \mid k?.b!]\!] & \mid & l'[\![k! \mid k?.a! \mid k?.b!]\!] & \mid\ l'?.a?.b?.c! \\
\rightarrow^* & \nu k.\ l'[\![\quad k?.a! \mid k?.b!]\!] & \mid & l'[\![\quad\ a! \mid\ \ b!]\!] & \mid\ l'?.a?.b?.c! \\
\rightarrow & \nu k. & & l'[\![\quad\ a! \mid\ \ b!]\!] & \mid\quad a?.b?.c! \rightarrow c!
\end{array}$$

Local ports can also be used to implement other standard programming constructs, such as various forms of recursion. Consider the operator $*(P)$ (omitted from our language) with reduction semantics $*(P) \rightarrow P \mid *(P)$. This operator can be encoded correctly using local ports and higher-order communication:

$$\mathsf{Rec}(P) \stackrel{\text{def}}{=} (\delta(p_{\text{L}}) \mid p_{\text{L}}!\lambda(P \mid \delta(p_{\text{L}})).\mathbf{0})\backslash p_{\text{L}} \quad \delta(p_{\text{L}}) \stackrel{\text{def}}{=} p_{\text{L}}?(x{:}\mathsf{Pr}).(\mathtt{run}\, x \mid p_{\text{L}}!x.\mathbf{0})$$

Again, an encoding $\mathsf{Rec}_{\mathtt{new}}$ using generated names would not be correct. The process $l[\![\mathsf{Rec}_{\mathtt{new}}(a!)]\!] \mid K_1 \mid l'?.a?.c!$ *can* reduce to $\nu k.l[\![\delta(k)]\!] \mid a?.c! \not\Downarrow_c$, but $l[\![*(a!)]\!] \mid K_1 \mid l'?.a?.c!$ cannot reduce to a system that does not have a barb on c.

Using only generated names, as in HOπPn [10] discussed in the introduction, internal choice and general recursion are not encodable ($*(P)$ is a primitive). □

Example 3.5. In the last example of this section we show that passivation of observer-generated code is observable; we will return to this example when motivating our LTS for HOPass (Thm. 4.1). Let $M_{3.5} = a?(x).*(l[\![\mathtt{run}\, x]\!])$ and $M'_{3.5} = a?(x).*(\mathtt{run}\, x \mid l!)$. In HOPass these two systems are distinguished by contextual equivalence. This is achieved by the context testing if instances of the code bound to x are passivated after an output on l. For example consider the context $K_{3.5} = [\cdot] \mid a!(\lambda b?.c!).b!.l?$; we have: $K_{3.5}[M_{3.5}] \rightarrow^* l[\![c!]\!] \mid *(l[\![\mathtt{run}(\lambda b?.c!)]\!]) \mid l? = N_{3.5} \not\Downarrow_b \Downarrow_c$. This can be matched by $K_{3.5}[M'_{3.5}]$ by performing at least the reductions $K_{3.5}[P'_{3.5}] \rightarrow^* c! \mid l! \mid *(\mathtt{run}(\lambda b?.c!) \mid l!) \mid l? = N'_{3.5} \not\Downarrow_b \Downarrow_c$. However, $N_{3.5} \not\cong_{\text{cxt}} N'_{3.5}$ because the passivation of l, $N_{3.5} \rightarrow *(l[\![\mathtt{run}(\lambda b?.c!)]\!]) \not\Downarrow_c$, cannot be matched by $N'_{3.5}$. □

4 From HOπ to HOPass

The goal of this paper is to give a first-order, symbolic LTS for HOPass in which standard weak bisimilarity fully captures contextual equivalence; i.e., weak

bisimulation is sound and complete with respect to ($\cong_{\text{cxt}}$). Moreover, we seek an LTS with a *small set of transitions* to simplify bisimulation proofs. In this section we motivate this new LTS by starting from a previous first-order LTS for a version of HOπ [6]—essentially HOPass without locations—and examining the additional observational power needed to add to the LTS in order to achieve a sound bisimulation for HOPass.

As in our previous work, there are two basic ingredients to this LTS. The first is the extension of the syntax with *symbolic higher-order inputs* that the observer provides to the system. We thus extend the syntax of processes with *symbolic agents* $\alpha \in \mathsf{Agent}$ representing observer-generated processes.

$$\mathcal{P} ::= \ldots \mid \alpha \qquad \mathsf{EProc} \qquad\qquad \mathcal{V} ::= \ldots \mid \lambda\mathcal{P} \qquad \mathsf{EValue}$$

The second ingredient in the construction of the LTS is the explicit recording of the knowledge of the observer interrogating a system using a *knowledge environment* Δ. Any value sent from the system to the observer is recorded in Δ; if this value is a code thunk it has a unique associated *output index* $\kappa \in \mathsf{OIdx}$. These constants provide an indirect way for referring to outputs and only appear in Δ and on transitions of the LTS; they do not appear in processes and values. We extend the freshness operator $\sharp$ to α's and κ's. A knowledge environment Δ has the following components:

1. $\text{names}(\Delta) \subset_{\mathsf{fin}} \mathsf{Name}$: a finite set of names known to the observer;
2. $\text{agents}(\Delta) \subset_{\mathsf{fin}} \mathsf{Agent}$: a finite set of observer-generated symbolic agents;
3. $\text{fun}(\Delta) \in \mathsf{OIdx} \to_{\mathsf{fin}} \mathsf{EValue}$: a finite function mapping output indices to system-generated code thunks.

Our LTS contains transitions over configurations of the form:

$$\mathcal{C} ::= \nu\widetilde{a}\langle \Delta \triangleright \mathcal{P} \rangle \qquad\qquad \mathsf{Conf}$$

The names in the vector $\widetilde{a}$ are generated by the system but are not known to the observer; they can appear in $\mathcal{P}$ and in the codomain of $\text{fun}(\Delta)$. We consider only configurations that are *well-formed*. That is, configurations $\nu\widetilde{a}\langle \Delta \triangleright \mathcal{P} \rangle$ recording all names and constants used in $\mathcal{P}$ and in the codomain of $\text{fun}(\Delta)$, whose names $\widetilde{a}$ are distinct pairwise and with respect to $\text{names}(\Delta)$. We also identify configurations up to alpha-renaming of $\widetilde{a}$.

As with our LTS for HOπ, our LTS for HOPass will contain one higher-order input rule in which the input value is a fresh $\lambda\alpha$ and two higher-order output rules that that extend $\text{fun}(\Delta)$ (one for output and one for passivation):

$$\begin{array}{llll}
\langle \Delta \triangleright c?(x{:}\mathsf{Pr}).\mathcal{P} \rangle \xrightarrow{c?\lambda\alpha} \langle \Delta, \alpha \triangleright \mathcal{P}\{\lambda\alpha/x\} \rangle & \text{if } \alpha \sharp \Delta & (\textsc{Tin-Pr}) \\
\langle \Delta \triangleright c!\lambda\mathcal{P}.\mathcal{Q} \rangle \xrightarrow{c!\kappa} \langle \Delta, \kappa{\mapsto}\lambda\mathcal{P} \triangleright \mathcal{Q} \rangle & \text{if } \kappa \sharp \Delta & (\textsc{Tout-Pr}) \\
\langle \Delta \triangleright c[\![\mathcal{P}]\!] \rangle \xrightarrow{c!\kappa} \langle \Delta, \kappa{\mapsto}\lambda\mathcal{P} \triangleright \mathbb{0} \rangle & \text{if } \kappa \sharp \Delta & (\textsc{Tpass})
\end{array}$$

The LTS also has fairly standard rules for internal steps, to propagate transitions over evaluation contexts, and for first-order input and output; we omit these

transitions in this extended abstract (see [6]). Transitions in this LTS are labelled either with τ (internal transition), with one of the I/O actions over generated names $\mu ::= c?n \mid c?\lambda\alpha \mid c!n \mid c!\kappa$, or, purely for producing internal transitions, with the corresponding I/O actions over local ports.

Two important transitions in the LTS for HOπ [6] are those implementing the notion of triggers. When a HOπ system runs a symbolic input $\lambda\alpha$ (removing the λ) it enables the transition

$$\nu\widetilde{a}\langle\Delta \triangleright \alpha\rangle \xrightarrow{\mathtt{run}\,\alpha} \nu\widetilde{a}\langle\Delta \triangleright \mathtt{0}\rangle \tag{1}$$

indicating the execution of some code within the observer. Conversely, the observer can run at any point system code stored under the index κ:

$$\nu\widetilde{a}\langle\Delta \triangleright \mathcal{P}\rangle \xrightarrow{\mathtt{run}\,\kappa} \nu\widetilde{a}\langle\Delta \triangleright \mathcal{P} \mid \mathtt{run}\,\Delta(\kappa)\rangle \tag{2}$$

Similarly, in HOPass we need to give the observer the ability to run system-generated code and detect the execution of observer code. However, the above two transitions are not adequate to give us soundness of weak bisimulation. In the rest of this section we give example *in*equivalent processes that can be distinguished by ($\cong_{\mathrm{cxt}}$) and motivate sufficient additions to the LTS of symbolic agents we have described so far. The following section contains the precise definitions of these additions and the relevant bisimulation.

We first show that (1) is no longer adequate in the presence of passivation. The observer should not just forget α after it has been run once; instead it requires the power to repeatedly *ping* α to ensure that the code implicitly represented by this symbolic agent is still alive and can communicate.

Example 4.1 (Example 3.5 revisited). Let us reconsider the systems $M_{3.5} = a?(x).*(l[\![\mathtt{run}\,x]\!])$. and $M'_{3.5} = a?(x).*(\mathtt{run}\,x \mid l!)$, which we have already seen are distinguished by ($\cong_{\mathrm{cxt}}$) using the context $K_{3.5} = [\cdot] \mid a!(\lambda b?.c!).b!.l?$ testing whether an output $c!$ is possible after the sequence of reductions $a?(\lambda b?.c!), b?, l?$.

The LTS transitions we have seen so far cannot perform such a test; $M_{3.5}$ and $M'_{3.5}$ are not distinguishable in the current LTS. Let us see how we might try to mimic the distinguishing tests performed by $K_{3.5}$. This context first sends in on the channel a the actual code $\lambda b?.c!$ but our transitions are only allowed to send in a symbolic agent $\lambda\alpha$. Next, $K_{3.5}$ communicates with the sent code on b—in the LTS this can only be translated to a transition of the form (1) above. Then, it passivates l and tries to communicate on c with the sent code. The passivation of l can be performed in the LTS, but we cannot translate the communication on c because the previous use of (1) has replaced α with $\mathtt{0}$.

What is required is the ability to repeatedly check if the symbolic agent α is still alive; i.e., that the system has not introduced a communication barrier between α and the observer by passivating the former. In our LTS for HOPass the single use transition (1) will be replaced by a more general *ping* transition $\alpha \rightsquigarrow \alpha'$ which "updates" an α at evaluation position within the system to a fresh α'. The observer can keep performing this indefinitely. In effect, in our new LTS α's represent the *state* of symbolic agents inside configurations, which

changes after an agent performs a transition. The transition $\alpha \rightsquigarrow \alpha'$ is a symbolic communication of an agent at state α with the observer, updating the state to a fresh α', which can be probed further (see Thm. 5.1). □

These *ping* transitions $\alpha \rightsquigarrow \alpha'$ discover *communication barriers* between symbolic agents and the external observer throughout the interrogation of a system. However, they are not sufficient to discover all of the intricacies of HOPass. Transparent locations together with passivation and reactivation can be used to create communication barriers *between* parts of the system, creating a form of *opaque locations*. We believe the ability to encode communication barriers captures the essence of location-dependent behaviour expressible in HOPass.

To crystallise this phenomenon we introduce a *trampoline* operator ($\bowtie$) which can be encoded in HOPass (as well as in HOπP [7]). Consider the process $P \bowtie Q$ with reduction rules:

$$P \bowtie Q \xrightarrow{\lambda} P' \bowtie Q \quad \text{if} \quad P \xrightarrow{\lambda} P' \qquad\qquad P \bowtie Q \xrightarrow{\lambda} P \bowtie Q' \quad \text{if} \quad Q \xrightarrow{\lambda} Q'$$

Essentially, $P \bowtie Q$ represents a communication barrier between the processes P and Q: they can communicate with their environment but not with each other. Provided $p_{\mathrm{L}}, {p_{\mathrm{L}}}' \sharp P, Q$, trampoline can be encoded in a fully-abstract manner:

$$P \bowtie Q \cong_{\mathrm{cxt}} \Big(p_{\mathrm{L}}[\![P]\!] \mid {p_{\mathrm{L}}}'!(\lambda Q).\mathbf{0} \mid *\big(p_{\mathrm{L}}?(x).{p_{\mathrm{L}}}'?(y).(p_{\mathrm{L}}[\![\mathtt{run}\, y]\!] \mid {p_{\mathrm{L}}}'!x.\mathbf{0})\big)\Big) \backslash p_{\mathrm{L}}, {p_{\mathrm{L}}}'$$

We prove an instance of this equivalence in Sect. 7 and use this operator extensively when motivating further symbolic transitions.

Example 4.2 (Communication barriers). Let us consider the systems $M_{4.2} = a?(x).(\mathtt{run}\, x \bowtie b!)$ and $M'_{4.2} = a?(x).(\mathtt{run}\, x \mid b!)$. These systems can be distinguished by $K_{4.2} = [\cdot] \mid a!(\lambda b?c!).\mathbf{0}$ because $K_{4.2}[M_{4.2}] \not\Downarrow_c$ whereas $K_{4.2}[M'_{4.2}] \Downarrow_c$. However, combinations of all LTS transitions we discussed so far are not able to distinguish them. We need to give agents the ability to communicate with the system, which in the next section we achieve by adding a transition $\mu/\alpha \rightsquigarrow \alpha'$ with which an agent transition $\alpha \rightsquigarrow \alpha'$ synchronises with a parallel action μ.

We also need a synchronisation transition between two running symbolic agents in order to observe the different behaviour of $N_{4.2} = a?(x).b?(y).(\mathtt{run}\, x \bowtie \mathtt{run}\, y)$ and $N'_{4.2} = a?(x).b?(y).(\mathtt{run}\, x \mid \mathtt{run}\, y)$. These are distinguished by $K'_{4.2} = [\cdot] \mid a!(\lambda c!).b!(\lambda c?.d!).\mathbf{0}$ since $K'_{4.2}[N_{4.2}] \not\Downarrow_d$ whereas $K'_{4.2}[N'_{4.2}] \Downarrow_d$. However, they are not distinguished by the transitions discussed so far. They will be distinguishable with a new synchronisation action $\alpha_1|\alpha_2 \rightsquigarrow \alpha'_1|\alpha'_2$, signalling that two agents α_1 and α_2 can communicate and become α'_1 and α'_2 (see Thm. 5.2). □

As we discussed, in HOπ the observer can use the transition $\mathtt{run}\, \kappa$ (2) to run system code indexed by κ in the knowledge environment of a configuration; this code runs in parallel with the system after the transition. Because again of the communication barriers encodable in HOPass, code run in parallel with the system may exhibit different behaviour than if it were run at the position of an agent; in the presence of passivation we need a more general symbolic transition.

Moreover, an adequate LTS for HOPass needs to enable the passivation of the entire system and any code executed by the observer. The former is motivated by the fact that ($\cong_{\text{cxt}}$) is closed under the context $a?(x).P \mid a![\cdot].\mathbf{0}$; thus, any related systems $M \cong_{\text{cxt}} M'$, possibly obtained after a number of reductions and context closures of ($\cong_{\text{cxt}}$), can be entirely passivated and reused as code thunks in P. The need for the latter is motivated in the next example.

To allow the observer to passivate running code we will use a set of special location names only as a namespace for observer-generated locations. We call these *abstract locations* $\gamma \in \mathsf{Aloc}$, and extend the syntax of HOPass once more to include such locations; we adjust ($\sharp$) to abstract locations and record γ's in knowledge environments: $\mathcal{P} ::= \ldots \mid \gamma[\![\mathcal{P}]\!]$. Intuitively, $\gamma[\![\mathcal{P}]\!]$ represents an agent that is currently running in a new location γ process $\mathcal{P}$, obtained from the observer's knowledge environment.

Remark 4.3. We use abstract locations instead of ordinary fresh names to *limit* the possible LTS transitions: these names need not be used as inputs or elsewhere in the system, considerably simplifying proofs of equivalence. □

Example 4.4 (Code execution and passivation). Consider the systems

$$M_{4.4} = \mathtt{new}\, t.a!(\lambda t?.c!).b?(x).(\mathtt{run}\, x \bowtie t!) \quad M'_{4.4} = \mathtt{new}\, t.a!(\lambda t?.c!).b?(x).(\mathtt{run}\, x \mid t!)$$

distinguished by the context $K_{4.4} = [\cdot] \mid a?(y).b!y.\mathbf{0}$ which simply relays a value from a to b. In $K_{4.4}[M_{4.4}]$ this leads to system $\nu t.((t?.c!) \bowtie t!) \not\Downarrow_c$; however, $K_{4.4}[M'_{4.4}]$ reduces to $\nu t.((t?.c!) \mid t!) \Downarrow_c$. Thus, this context distinguishes the two systems by running $(\lambda t?.c!)$ at the only position of $M_{4.4}$ where communication with the $t!$ is impossible (in the LHS of the $\bowtie$). Contexts that do not cause the execution of $(\lambda t?.c!)$ at that position cannot distinguish $M_{4.4}$ from $M'_{4.4}$. The transitions we have discussed so far encode observations made by such contexts and therefore fail to distinguish the two systems.

To see that, we consider the interrogation of a configuration where $\lambda M_{4.4}$ is in Δ, from which the observer, using the previously discussed transitions, can only reach configurations of the form $\nu\widetilde{t}\langle \Delta \triangleright \prod \alpha_i \bowtie t_i! \rangle$, with $\Delta(\kappa_i) = \lambda t_i?.c!$. The only way for the observer to run a κ_i *without* enabling a communication on t_i (and thus an observable $c!$ transition) is to run κ_i at α_i (as the context $K_{4.4}$ above did). The preceding LTS transitions do not capture such a move. Therefore we introduce a separate transition $\alpha \rightsquigarrow \gamma[\![\kappa]\!]$ which replaces a symbolic agent α with $\gamma[\![\Delta(\kappa)]\!]$, for a fresh γ.

Now consider the systems:

$$N_{4.4} = \mathtt{new}\, t.a!(\lambda t?c!).b!(\lambda t!).\mathbf{0} \qquad N'_{4.4} = \mathtt{new}\, t.a!(\lambda t!).b!(\lambda t?c!).\mathbf{0}$$

To distinguish them, the observer needs to input both code thunks on a and b, run both, and passivate one of them after they communicate on t. The last move is not possible with the transitions we have seen so far. A context that performs this scenario and distinguishes $N_{4.4}$ from $N'_{4.4}$ is $K'_{4.4} = [\cdot] \mid a?(x).b?(y).(l[\![\mathtt{run}\, x]\!] \mid \mathtt{run}\, y \mid l?.c?.d!)$. We have $K'_{4.4}[N_{4.4}] \not\Downarrow_d$ but $K'_{4.4}[N'_{4.4}] \Downarrow_d$.

In our new LTS, transitions $a!\kappa_1, b!\kappa_2, \alpha_1 \rightsquigarrow \gamma_1[\![\kappa_1]\!], \alpha_2 \rightsquigarrow \gamma_2[\![\kappa_2]\!]$ let the observer receive and run the code emitted from the systems (provided there are

$$\textsc{Trun-}\kappa\ \frac{\Delta(\kappa) = \lambda\mathcal{P} \qquad \gamma \sharp \Delta}{\langle \Delta \triangleright \alpha \rangle \xrightarrow{\alpha \rightsquigarrow \gamma[\![\kappa]\!]} \langle \Delta, \gamma \triangleright \gamma[\![\mathcal{P}]\!] \rangle} \qquad \textsc{Tpass-}\gamma\ \frac{\kappa, \alpha \sharp \Delta}{\langle \Delta \triangleright \gamma[\![\mathcal{P}]\!] \rangle \xrightarrow{\gamma[\![\kappa]\!] \rightsquigarrow \alpha} \langle \Delta, \alpha, \kappa \mapsto \lambda\mathcal{P} \triangleright \alpha \rangle}$$

$$\textsc{TioL@}\alpha\ \frac{\langle \Delta \triangleright \mathcal{P} \rangle \xrightarrow{\alpha_1 \rightsquigarrow \alpha_2} \langle \Delta' \triangleright \mathcal{P}' \rangle \quad \langle \Delta' \triangleright \mathcal{Q} \rangle \xrightarrow{\mu} \langle \Delta'' \triangleright \mathcal{Q}' \rangle}{\langle \Delta \triangleright \mathcal{P} \mid \mathcal{Q} \rangle \xrightarrow{\mu/\alpha_1 \rightsquigarrow \alpha_2} \langle \Delta'' \triangleright \mathcal{P}' \mid \mathcal{Q}' \rangle} \qquad \textsc{Tsync}\ \frac{\langle \Delta \triangleright \mathcal{P} \rangle \xrightarrow{\alpha_1 \rightsquigarrow \alpha_2} \langle \Delta' \triangleright \mathcal{P}' \rangle \quad \langle \Delta' \triangleright \mathcal{Q} \rangle \xrightarrow{\alpha_3 \rightsquigarrow \alpha_4} \langle \Delta'' \triangleright \mathcal{Q}' \rangle}{\langle \Delta \triangleright \mathcal{P} \mid \mathcal{Q} \rangle \xrightarrow{\alpha_1 | \alpha_3 \rightsquigarrow \alpha_2 | \alpha_4} \langle \Delta'' \triangleright \mathcal{P}' \mid \mathcal{Q}' \rangle}$$

$$\textsc{Tsig}\ \frac{\alpha_2 \sharp \Delta}{\langle \Delta \triangleright \alpha_1 \rangle \xrightarrow{\alpha_1 \rightsquigarrow \alpha_2} \langle \Delta, \alpha_2 \triangleright \alpha_2 \rangle} \qquad \textsc{Textr@}\alpha\ \frac{\nu\widetilde{a}\langle \Delta, n \triangleright \mathcal{P} \rangle \xrightarrow{c!n/\alpha_1 \rightsquigarrow \alpha_2} \nu\widetilde{b}\langle \Delta' \triangleright \mathcal{P}' \rangle \quad c \neq n}{\nu n, \widetilde{a}\langle \Delta \triangleright \mathcal{P} \rangle \xrightarrow{c!n/\alpha_1 \rightsquigarrow \alpha_2} \nu\widetilde{b}\langle \Delta' \triangleright \mathcal{P}' \rangle}$$

Fig. 3. LTS: symbolic agent transitions (omitting symmetric rules)

running symbolic agents α_1 and α_2). However, to enable further passivation of this code, we introduce transitions of the form $\gamma_1[\![\kappa_{\mathrm{fr}}]\!] \rightsquigarrow \alpha_{\mathrm{fr}}$ which let the observer passivate the code running in γ_1, replacing it with a fresh symbolic agent α_{fr}, and indexing it by a fresh κ_{fr} in the knowledge environment. With the addition of this last LTS transition the observer can distinguish the above systems in the same way as $K'_{4.4}$ does (see Thm. 5.3 for details). □

5 First-Order Symbolic Agent Transitions

The previous section briefly described internal (τ) and communication (μ) transitions and focused on motivating a set of new symbolic (ζ) transitions for an adequate LTS for HOPass. Here we give the precise rules of the new ζ-transitions, all of which describe a *limited symbolic execution* of agents running in a configuration. An agent is an observer-generated process represented simply by $\alpha \in \mathsf{PConst}$, or an abstract location $\gamma[\![\mathcal{P}]\!]$ running a single system-generated code thunk ($\gamma \in \mathsf{LConst}$). The result is an LTS with *first-order* transitions ($\eta ::= \tau \mid \mu \mid \zeta$) simplifying bisimulation proofs.

Observer transitions are generated by the rules shown in Fig. 3 and are annotated with one of the following labels:

$$\zeta \ ::= \ \alpha \rightsquigarrow \alpha \ \mid \ \mu/\alpha \rightsquigarrow \alpha \ \mid \ \alpha|\alpha \rightsquigarrow \alpha|\alpha \ \mid \ \alpha \rightsquigarrow \gamma[\![\kappa]\!] \ \mid \ \gamma[\![\kappa]\!] \rightsquigarrow \alpha$$

These transitions encode symbolic moves performed by agents, visible to the overall observer, in order to *reconfigure* agents and interrogate the system. They fall naturally into two groups, the first concerned with *communication barriers* and the second with code *execution* and *passivation*. The first three involve communication barriers.

Ping: $\alpha_1 \rightsquigarrow \alpha_2$. This transition, produced by rule Tsig, allows the observer to determine if an agent α_1 is running. There is a communication barrier between the observer and α_1 only if α_1 is not running in the configuration. As a result of

this transition, a running instance of α_1 is replaced by a fresh α_2, distinguishing it from other instances of α_1 in the configuration. In this way this transition can be used to distinguish the processes in Thm. 4.1.

Example 5.1 (Thm. 4.1 continued). Let us see how $M_{3.5}$ and $M'_{3.5}$ can be distinguished. Consider an observer examining configurations such as $\langle \Delta \triangleright M_{3.5} \rangle$ and $\langle \Delta \triangleright M'_{3.5} \rangle$, where $\Delta = \{a, l\}$. After a transition $a?\lambda\alpha_1$ we get the configurations $\langle \Delta, \alpha_1 \triangleright *(l[\![\mathtt{run}\,\lambda\,\alpha_1]\!]) \rangle$ and $\langle \Delta, \alpha_1 \triangleright *(\mathtt{run}\,\lambda\,\alpha_1 \mid l!) \rangle$. After τ-transitions and transitions $\alpha_1 \rightsquigarrow \alpha_2$ and $l!\kappa$ we get $\langle \Delta, \widetilde{\alpha}, \kappa \mapsto \lambda\alpha_2 \triangleright *(l[\![\mathtt{run}\,\lambda\,\alpha_1]\!]) \rangle$ and $\langle \Delta, \widetilde{\alpha}, \kappa \mapsto \lambda\mathtt{0} \triangleright \alpha_2 \mid *(\mathtt{run}\,\lambda\,\alpha_1 \mid l!) \rangle$. The latter configuration has an $\alpha_2 \rightsquigarrow \alpha_3$ transition but the former does not. □

I/O from a Symbolic Agent: $\mu/\alpha_1 \rightsquigarrow \alpha_2$. Because of the communication barriers encodable in HOPass, an agent may or may not be running at the same time as an observable μ action in a configuration. This transition, due to rule TioL@α and its symmetric one, allows the observer to detect this situation and distinguish systems $M_{4.2}$ and $M'_{4.2}$ in Thm. 4.2. As with standard name output, a name output detected by an agent can extrude a private name, moving it from the list of bound names into the knowledge environment (rule TextR@α). Note the chaining of the knowledge environments in the two premises of the rule that accumulates the effects of the two transitions in the final Δ''.

Agent Synchronisation: $\alpha_1|\alpha_3 \rightsquigarrow \alpha_2|\alpha_4$. For the same reason as above, this transition allows the observer to detect whether two symbolic agents are simultaneously running and can thus communicate. As before, the effects of the two transitions in the premises (i.e., the extension of Δ with fresh α_2 and α_4) are accumulated in the final Δ'' by chaining. Such a transition, generated by Tsync, can be used to distinguish the systems $N_{4.2}$ and $N'_{4.2}$ in Thm. 4.2.

Example 5.2 (Thm. 4.2 continued). An observer can distinguish configurations such as $\langle \Delta \triangleright M_{4.2} \rangle$ and $\langle \Delta \triangleright M'_{4.2} \rangle$ because the latter can perform the transition sequence $a?\alpha_1, b!/\alpha_1 \rightsquigarrow \alpha_2$ but the former cannot. Similarly, the observer can distinguish $\langle \Delta \triangleright N_{4.2} \rangle$ from $\langle \Delta \triangleright N'_{4.2} \rangle$ because the latter can perform the transition sequence $a?\alpha_1, b?\alpha_3, \alpha_1|\alpha_3 \rightsquigarrow \alpha_2|\alpha_4$ but the former cannot. □

We now detail the symbolic transitions concerned with code execution and passivation. These only use fresh abstract locations γ, not generated names, simplifying the LTS and the construction of witness bisimulations. We also ensure that at each abstract location γ only one system-generated process is executing at any time, further simplifying the LTS.

Code Execution: $\alpha \rightsquigarrow \gamma[\![\kappa]\!]$. With this transition (due to Trun-κ) the observer sends a system-generated code thunk, indexed in the knowledge environment by κ, to the location of a running agent α to be executed. The agent originates from a higher-order input transition (rule TIn-Pr in Sect. 4) and the thunk from a higher-order system output (rule Tout-Pr), before or after the input. After the transition, α is replaced by a fresh abstract location γ in which the code in $\Delta(\kappa)$ runs (and only that).

Abstract Location Passivation: $\gamma[\![\kappa]\!] \rightsquigarrow \alpha$**.** This transition (due to TPASS-γ) allows the observer to passivate an abstract location γ, which has previously been introduced by the symbolic transition just described, $\alpha \rightsquigarrow \gamma[\![\kappa]\!]$.

Example 5.3 (Thm. 4.4 revisited). We have seen that systems $N_{4.4}$ and $N'_{4.4}$ are not contextually equivalent. Here we show how an observer can use the above two symbolic transitions to distinguish them, when examining the configurations $\langle \Delta_1 \triangleright \alpha_1 \mid \alpha_2 \rangle$ and $\langle \Delta'_1 \triangleright \alpha_1 \mid \alpha_2 \rangle$, where $\Delta_1 = \Delta, \kappa \mapsto \lambda N_{4.4}$, $\Delta'_1 = \Delta, \kappa \mapsto \lambda N'_{4.4}$, and $\Delta = \{a, b, c, \widetilde{\alpha}\}$. After transition $\alpha_1 \rightsquigarrow \gamma_1[\![\kappa]\!]$ we get

$$\nu t\langle \Delta_1, \gamma_1 \triangleright \gamma_1[\![N_{4.4}]\!] \mid \alpha_2 \rangle \qquad \nu t\langle \Delta'_1, \gamma_1 \triangleright \gamma_1[\![N'_{4.4}]\!] \mid \alpha_2 \rangle$$

and after a sequence of weak transitions $a!\kappa_1, b!\kappa_2, \gamma_3[\![\kappa_3]\!] \rightsquigarrow \alpha_3$:

$$\nu t\langle \Delta_{\kappa_1,\kappa_2} \triangleright \alpha_3 \mid \alpha_2 \rangle \qquad \nu t\langle \Delta_{\kappa_2,\kappa_1} \triangleright \alpha_3 \mid \alpha_2 \rangle$$

where $\Delta_{x,y} = \Delta_1, \widetilde{\alpha}, \widetilde{\gamma}, (x \mapsto \lambda t?c!), (y \mapsto \lambda t!), (\kappa_3 \mapsto \lambda \mathbf{0})$. The observer can now run both κ_1 and κ_2 (with the transitions $\alpha_3 \rightsquigarrow \gamma_3[\![\kappa_1]\!]$ and $\alpha_2 \rightsquigarrow \gamma_2[\![\kappa_2]\!]$) and obtain:

$$\nu t\langle \Delta_{\kappa_1,\kappa_2}, \widetilde{\gamma} \triangleright \gamma_3[\![t?c!]\!] \mid \gamma_2[\![t!]\!] \rangle \qquad \nu t\langle \Delta_{\kappa_2,\kappa_1}, \widetilde{l} \triangleright \gamma_3[\![t!]\!] \mid \gamma_2[\![t?c!]\!] \rangle$$

Only the left configuration can now perform a weak sequence of transitions $(\gamma_2[\![\kappa_3]\!] \rightsquigarrow \alpha_4), c!$. Hence the original systems are differentiated by the observer. Note that the passivation of γ_1 in this example re-introduced a new symbolic agent α_3 in its place, allowing the observer to continue the interrogation of the configuration. This is why TPASS-γ introduces a new constant in our LTS. □

6 Weak Bisimulation Theory

We employ the standard bisimulation theory, applied to the LTS of configurations generated by our first-order agent semantics outlined in the previous sections. This is then restricted to a *subset* of configurations. Weak bisimilarity over this subset is sound and complete with respect to contextual equivalence (Thm. 3.2). We use the standard notation ($\stackrel{\eta}{\Rightarrow}$) to mean the reflexive transitive closure of ($\stackrel{\tau}{\rightarrow}$), when $\eta = \tau$, and ($\stackrel{\tau}{\Rightarrow}\stackrel{\eta}{\rightarrow}\stackrel{\tau}{\Rightarrow}$) otherwise.

Definition 6.1 (Weak Bisimulation). $\mathbb{R}$: Conf $\times$ Conf *is a weak bisimulation when for all* $\mathcal{C}_1 \mathbb{R} \mathcal{C}'_1$ *the following condition and its converse are satisfied: If* $\mathcal{C}_1 \stackrel{\eta}{\rightarrow} \mathcal{C}_2$ *then there exists* $\mathcal{C}'_2$ *such that* $\mathcal{C}'_1 \stackrel{\eta}{\Rightarrow} \mathcal{C}'_2$ *and* $\mathcal{C}'_1 \mathbb{R} \mathcal{C}'_2$.

The largest weak bisimulation, *weak bisimilarity* ($\approx$), is the union of all weak bisimulations; it is straightforward to show that this is an equivalence relation.

We have deliberately restricted the number and form of symbolic transitions, so as to facilitate the description of witness bisimulations when proving systems equivalent. For example there is no direct way in which the observer can execute at top-level code received from the system, indexed by a κ; this was even necessary in the simpler language of HOπ [6]. In the current framework, the observer interrogating a system, needs to have already executing within the system symbolic agents, represented either by occurrences of α's or γ's. Because of this, we

let the observer interrogate a system $\nu\widetilde{a}.P$ by transitions in the agent LTS by starting in the configuration $\nu\widetilde{a}\langle\Delta, \kappa{\mapsto}\lambda P \triangleright \prod \alpha_i\rangle$. Here each symbolic agent α_i allows the observer to initiate the interrogation, by executing one of the symbolic actions from Fig. 3. In fact only *two* such symbolic agents are necessary.

Theorem 6.2 (Soundness and Completeness of $(\approx)$). *Let $M = \nu\widetilde{m}.P$, $N = \nu\widetilde{n}.Q$ be closed systems. Then for $\Delta = \{\widetilde{c}, \widetilde{\alpha}\} \supseteq \mathrm{fn}(M), \mathrm{fn}(N), \alpha_1, \alpha_2$: $M \cong_{\mathrm{cxt}} N$ iff $\nu\widetilde{m}\langle\Delta, \kappa{\mapsto}\lambda P \triangleright \alpha_1 \mid \alpha_2\rangle \approx \nu\widetilde{n}\langle\Delta, \kappa{\mapsto}\lambda Q \triangleright \alpha_1 \mid \alpha_2\rangle$.*

7 Example Equivalence

The encoding in HOPass of internal choice $(\oplus)$, replication $(*(\,))$, and the trampoline operator $(\bowtie)$ are fully abstract. Here we prove an instance of the last: in HOPass extended with $(\bowtie)$ and replication, $M = a?(x,y).(\mathtt{run}\, x \bowtie \mathtt{run}\, y) \cong_{\mathrm{cxt}} a?(x,y).(\mathtt{run}\, x \bowtie_{\mathsf{enc}} \mathtt{run}\, y) = M'$, where $(\bowtie_{\mathsf{enc}})$ is the encoding on page 175. Soundness of $(\approx)$ holds for the extended language. Thus, from Thm. 6.2, it suffices to show $\langle\Delta, \kappa{\mapsto}\lambda M \triangleright \alpha_1 \mid \alpha_2\rangle \approx \langle\Delta, \kappa{\mapsto}\lambda M' \triangleright \alpha_1 \mid \alpha_2\rangle$. To reduce the size of the proof we make use of a standard *up-to beta steps* technique, similar to that in our previous work for HOπ [6], and observe that internal `run` transitions and all transitions involving communication on p_{L} and ${p_{\mathrm{L}}}'$ in $(\bowtie_{\mathsf{enc}})$ are beta steps. We construct the following relation on well-formed configurations and prove it is a weak bisimulation up to beta steps by induction on the construction and enumeration of the possible LTS transitions.

$$\begin{array}{c}\langle\Delta, \widetilde{\kappa}_1{\mapsto}\lambda M, \kappa_{21}{\mapsto}\lambda\alpha_{11} \bowtie \alpha_{12}, \ldots \kappa_{2m}{\mapsto}\lambda\alpha_{1m} \bowtie \alpha_{2m} \triangleright \alpha_3 \mid \alpha_4\rangle \\ \mathbb{R}\ \langle\Delta, \widetilde{\kappa}_1{\mapsto}\lambda M', \kappa_{21}{\mapsto}\lambda\alpha_{11} \bowtie_{\mathsf{enc}} \alpha_{12}, \ldots \kappa_{2m}{\mapsto}\lambda\alpha_{1m} \bowtie_{\mathsf{enc}} \alpha_{2m} \triangleright \alpha_3 \mid \alpha_4\rangle \\ (\mathcal{C}, \gamma)\{\{\gamma[\![\mathtt{run}\, \mathcal{C}(\kappa)/\alpha]\!]\}\}\ \mathbb{R}\ (\mathcal{C}', \gamma)\{\{\gamma[\![\mathtt{run}\, \mathcal{C}'(\kappa)/\alpha]\!]\}\}\ \text{if}\ \mathcal{C}\ \mathbb{R}\ \mathcal{C}' \\ (\mathcal{C}, \widetilde{\alpha})\{\{\alpha_1 \bowtie \alpha_2/M\}\}\ \mathbb{R}\ (\mathcal{C}', \widetilde{\alpha})\{\{\alpha_1 \bowtie_{\mathsf{enc}} \alpha_2/M\}\}\ \text{if}\ \mathcal{C}\ \mathbb{R}\ \mathcal{C}'\end{array}$$

Here $\{\{\mathcal{P}/\mathcal{Q}\}\}$ replaces *one* occurrence of $\mathcal{Q}$ with $\mathcal{P}$ in a configuration, and $(\mathcal{C}, \Delta)$ extends the knowledge environment of $\mathcal{C}$ with a fresh Δ; $\mathcal{C}(\kappa)$ denotes the code indexed by κ in the environment of $\mathcal{C}$. In the base case of the construction, related knowledge environments contain an arbitrary number of indices to the initial systems ($\kappa_{1i}{\mapsto}\lambda M$ and $\kappa_{1i}{\mapsto}\lambda M'$), as well as an arbitrary number of $\kappa_{2i}{\mapsto}\lambda\alpha_{1i} \bowtie \alpha_{2i}$ and $\kappa_{2i}{\mapsto}\lambda\alpha_{1i} \bowtie_{\mathsf{enc}} \alpha_{2i}$, where the α_{1i}'s are not necessarily pairwise distinct (similarly for the α_{2i}'s).

8 Conclusions

We presented the first first-order bisimulation proof technique for a distributed language with passivation and private names, which is sound and complete with respect to weak barbed congruence, the contextual equivalence associated with weak bisimulation. In our language, code behaviour is location-dependent, the usual encodings of useful systems are possible, and unnecessary complexities with free names are avoided. We believe that our technique can be adapted to other distributed languages with location-dependent behaviour, and indeed to HOπP [7], a passivation language where free names are observable for which the only

available proof technique is context bisimulation. Normal and environmental bisimulation are sound only for sublanguages of HOπP [7, 9], and the latter technique is sound and complete for a language with generative names [10]. These language variations, however, cannot express many useful systems, such as those with internal choice, replication, or higher-order values containing input processes. In other languages with passivation, context bisimulation is only sound for the weak case [2, 3] or sound and complete for only the strong case [14]. Unlike context and environmental bisimulation, our proof technique avoids any universal quantification over contexts. This is achieved by a labelled transition system in which higher-order input values are replaced by abstract agents which can perform limited symbolic transitions within systems, simplifying proofs.

References

1. Amadio, R.M., Dam, M.: Reasoning about higher-order processes. In: Mosses, P.D., Nielsen, M. (eds.) CAAP 1995, FASE 1995, and TAPSOFT 1995. LNCS, vol. 915, pp. 202–216. Springer, Heidelberg (1995)
2. Castagna, G., Vitek, J., Zappa Nardelli, F.: The Seal calculus. Information and Computation 201(1), 1–54 (2005)
3. Godskesen, J.C., Hildebrandt, T.: Extending Howes method to early bisimulations for typed mobile embedded resources with local names. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 140–151. Springer, Heidelberg (2005)
4. Jeffrey, A., Rathke, J.: Contextual equivalence for higher-order pi-calculus revisited. LMCS 1(1:4) (2005)
5. Koutavas, V., Hennessy, M.: A testing theory for a higher-order cryptographic language. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 358–377. Springer, Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-19718-5_19
6. Koutavas, V., Hennessy, M.: First-order reasoning for higher-order concurrency. Computer Languages, Systems & Structures 38(3), 242–277 (2012)
7. Lenglet, S.: Schmitt A, and Stefani J.-B. Characterizing contextual equivalence in calculi with passivation. Information and Computation 209(11), 1390–1433 (2011)
8. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 685–695. Springer, Heidelberg (1992)
9. Piérard, A., Sumii, E.: Sound bisimulations for higher-order distributed process calculus. In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 123–137. Springer, Heidelberg (2011)
10. Pierard, A., Sumii, E.: A higher-order distributed calculus with name creation. In: LICS, pp. 531–540. IEEE Computer Society (June 2012)
11. Sangiorgi, D.: Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms. PhD thesis, University of Edinburgh (1992)
12. Sangiorgi, D.: From pi-calculus to higher-order pi-calculus–and back. In: Gaudel, M.-C., Jouannaud, J.-P. (eds.) CAAP 1993, FASE 1993, and TAPSOFT 1993. LNCS, vol. 668, pp. 151–166. Springer, Heidelberg (1993)
13. Sangiorgi, D.: On the bisimulation proof method. MSCS 8(5), 447–479 (1998)
14. Schmitt, A., Stefani, J.-B.: The Kell calculus: A family of higher-order distributed process calculi. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 146–178. Springer, Heidelberg (2005)
15. Vivas, J.-L., Dam, M.: From higher-order π-calculus to π-calculus in the presence of static operators. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 115–130. Springer, Heidelberg (1998)

A Theory of Name Boundedness

Reiner Hüchting[1], Rupak Majumdar[2], and Roland Meyer[1]

[1] University of Kaiserslautern
[2] MPI-SWS

Abstract. We develop a theory of *name-bounded π-calculus processes*, which have a bound on the number of restricted names that holds for all reachable processes. Name boundedness reflects resource constraints in practical reconfigurable systems, like available communication channels in networks and address space limitations in software.

Our focus is on the algorithmic analysis of name-bounded processes. First, we provide an extension of the Karp-Miller construction that terminates and computes the coverability set for any name-bounded process. Moreover, the Karp-Miller tree shows that name-bounded processes have a pumping bound as follows. When a restricted name is distributed to a number of sequential processes that exceeds this bound, the name may be distributed arbitrarily. Second, using the bound, we construct a Petri net bisimilar to the name-bounded process. The Petri net keeps a reference count for each restricted name, and recycles names that are no longer in use. The pumping property ensures that bounded zero tests are sufficient for recycling. With this construction, name-bounded processes inherit decidability properties of Petri nets. In particular, reachability is decidable for them. We complement our decidability results by a non-primitive recursive lower bound.

1 Introduction

The π-calculus is an established formalism for modeling and reasoning about reconfigurable systems that dynamically create and destroy communication links at runtime. While Turing complete in general, there are interesting subclasses of π-calculus where important verification problems remain decidable. In this paper, we propose and investigate a natural subclass of π-calculus processes: *name-bounded processes*. These are processes that have a bound on the number of restricted names that holds for all reachable processes.

Name-bounded processes are interesting for various reasons. First, from a theoretical perspective, they form a natural subclass of π-calculus that is expressive enough to subsume many other classes, but for which, as we will show, analysis questions remain decidable. Second, from a practical perspective, name boundedness captures natural constraints on an implementation that limit the total number of physical resources used at any point without constraining the number of processes. For example, a networked system can use a limited number of IP addresses or communication ports, but allow many clients to multiplex these resources. Finally, name boundedness is useful in verification as an approximate model for Turing-complete systems. For example, a finite abstraction of the shared heap in a concurrent program leads to a name-bounded model.

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 182–196, 2013.

Motivated by these applications in modeling, synthesis, and verification, we focus on the algorithmic analysis of name-bounded systems. Coverability is decidable for the model using generic well-structuredness arguments. Our main contributions, listed below, show effectiveness and decidability results beyond coverability. We also outline the theoretical tools we develop for the proofs.

Contribution 1: Synthesis Given an *arbitrary* π-calculus process P and a bound $b \in \mathbb{N}$, we prove it decidable to check whether P is name-bounded by b. Phrased differently, given a system P and a resource constraint b on the execution environment, one can decide if all executions of P use at most b resources. Note that the result is non-trivial since 0-bounded systems may already have an infinite state space (they can simulate Petri nets). It should rather be contrasted with the general name boundedness problem, which of course is undecidable.
Tool 1: Karp and Miller algorithm Behind this result is a novel extension of the Karp and Miller construction [9]. Given a process, our algorithm computes a finite representation of the downward closure of the reachability set; and the computation terminates precisely for name-bounded processes. Extending the Karp and Miller construction for Petri nets to name-bounded processes is non-trivial, since it is not sufficient to merely compare two limit elements (as for Petri nets). We extend the construction by tracking names instantiated in intermediary transitions to decide when a transition sequence can be accelerated.

Contribution 2: Petri net construction As second contribution we show that every name-bounded processes can be translated into a bisimilar Petri net. As a result, even subtle verification problems like reachability are decidable for name-bounded systems.
Tool 2: Pumping constant The idea behind the Petri net construction is to provide a finite set of instances $(a, 0), \ldots, (a, b-1)$ that can be used to represent the restricted name a. If we encounter a restriction νa in a run, we represent it by an instance that is not present in the current process. To check that an instance is not used, we keep a reference count. The instance is not used if and only if this count is zero.

Unfortunately, Petri nets cannot perform zero-tests. Instead, we show that the reference counts can be bounded using a pumping constant $p \in \mathbb{N}$ such that once an instance (a, k) is known to more than p processes, it can be distributed to arbitrarily many processes. This has the following consequence. If the reference counter of an instance (a, k) exceeds p, the instance cannot be reused. Phrased differently, we only need to keep the precise reference count up to p. This bound allows us to implement zero-tests with a Petri net. We show that the pumping constant can be computed from the Karp-Miller tree.

Contribution 3: Lower bound We show that verification problems for name-bounded processes, be it coverability or reachability, have non-primitive recursive space complexity. In contrast, reachability for the related model of Petri nets is only known to be EXPSPACE-hard [10], and coverability is EXPSPACE-complete [18]. Moreover, most known extensions of Petri nets with non-primitive recursive lower bounds for coverability have undecidable reachability problems.

Tool 3: Space-bounded Turing machine simulation To establish the lower bound result, we show how to simulate Turing machines where the tape-size is bounded by the Ackermann-function $A(n)$ with name-bounded processes. Our construction combines a classical result for Petri nets with the expressiveness of π-calculus as follows. With the construction of Mayr and Meyer [11], we obtain a process that generates up to $A(n)$ waiting processes and terminates. We then use a π-calculus modeling trick to arrange these processes into a list representing the Turing tape, and simulate the machine by communicating the state of the machine between tape cells.

Thus, our results resolve —positively— the algorithmic landscape of a natural and expressive fragment of the π-calculus.

Related Work. We recall the translations of reconfigurable systems into place/transition Petri nets that have been proposed in the literature. None of them can handle the name-bounded processes we consider here. Translations for restriction-bounded processes (where restrictions do not occur inside recursion) can be found in [14,3]. These works propose an identity-aware semantics similar to the one we use here, but do not introduce reference counters and pumping constants. The reason we need these tools is the generality of name-bounded processes. Our new class strictly subsumes restriction-bounded and, at the same time, finite control processes (FCPs) [5]. We are thus faced with unbounded parallelism combined with unboundedly many names. FCPs are translated into polynomial-sized safe Petri nets in [15]. We show that such a compact encoding cannot exist for name-bounded processes. In the worst case, the Petri nets have to be non-primitive recursive. The structural translation [13] identifies groups of processes that share restricted names. It yields a finite representation precisely for the class of structurally stationary processes, which are incomparable with name-bounded processes.

There are alternative translations of reconfigurable systems into higher-level Petri nets. Due to the expressive target formalism, it is not possible to deduce decidability results for properties like reachability from them. The class in [1] restricts the processes that can receive on a generated name. A translation into transfer nets yields decidability of control reachability. A translation of π-calculus into high-level Petri nets is given in [6]. In [17], FCPs are encoded into history dependent automata where states are labelled by names to represent restrictions.

Decidability of reachability in related process models, such as a fragment of mobile ambients [4] and a variant of CCS [8], has been shown by reduction to Petri net reachability. Unlike these papers, our construction gives a stronger correspondence: a Petri net that is bisimilar to a name-bounded process.

Name-bounded processes are bounded in depth in the sense that the nesting of restrictions is limited [12]. Therefore, positive results for depth-bounded systems, like decidability of coverability [20], carry over to name boundedness. However, reachability is undecidable for depth-bounded systems [14], and the coverability set is not computable [2]. This motivated our search for subclasses of depth boundedness and we show here that both problems, reachability and coverability

set computation, can be solved for name-bounded models. Further, we can decide whether an arbitrary process is name-bounded by a given $b \in \mathbb{N}$.

Our results rely on an adaptation of the Karp-Miller construction [9]. Finkel and Goubault-Larrecq [7] extend the same algorithm to compute coverability sets for general well-structured systems. There are two reasons why we propose a specific variant. First, we want our algorithm to be sound and complete for general, Turing-complete systems, but terminate only for name-bounded ones. This is needed to decide name boundedness when $b \in \mathbb{N}$ is given. The approach in [7] is always guaranteed to terminate, and hence does not handle Turing-complete models. Second, soundness requires an acceleration that depends on the labels along the path. It is unclear how to encode this into [7].

2 Name-bounded Processes

The π-Calculus. We recall the basics on π-calculus [16,19]. The π-calculus encodes computation using processes that exchange messages over channels. Messages and channels are untyped: a message that is received may serve as channel in further interactions. Formally, messages and channels are *names* a, b, x, y from a countable set of names $\mathcal{N}$. Processes communicate by synchronizing on *prefixes* π that are *sending* $\overline{x}\langle y\rangle$ or *receiving* $x(z)$. From these elementary communications, we build models of reconfigurable systems using non-deterministic choice $+$, parallel composition $\mid$, restriction νa, and parameterized recursion $K\lfloor\tilde{a}\rfloor$. To implement recursion, we introduce process identifiers, ranging over K, together with defining equations $K(\tilde{x}) := P$, where P is again a process and $\tilde{x}$ is a sequence of distinct names so that $|\tilde{a}| = |\tilde{x}|$. Every process relies on finitely many defining equations.

Formally, *processes* P, Q, R from the set of processes $\mathcal{P}$ are defined by

$$M ::= \mathbf{0} \mid \pi.P \mid M_1 + M_2 \qquad\qquad P ::= M \mid K\lfloor\tilde{a}\rfloor \mid P_1 \mid P_2 \mid \nu a.P$$

Processes M and $K\lfloor\tilde{a}\rfloor$ are called *sequential* as they are the basic building blocks for parallel compositions. We also use syntax S to indicate that the given process is sequential, and write $\mathcal{S}$ for the set of all sequential processes. We abbreviate k-fold parallel compositions of the same process P by P^k, where $P^0 := \mathbf{0}$.

The intended semantics of a call $K\lfloor\tilde{a}\rfloor$ is that the process behaves like P, but with $\tilde{x}$ replaced by $\tilde{a}$. This replacement is formalized by the application of *substitutions*. A substitution $\{\tilde{a}/\tilde{x}\}$ is a function from $\mathcal{N}$ to $\mathcal{N}$ that maps $\tilde{x}$ to $\tilde{a}$ and leaves all names outside $\tilde{x}$ unchanged. The application of $\{\tilde{a}/\tilde{x}\}$ to process P is denoted by $P\{\tilde{a}/\tilde{x}\}$ and defined in the standard way [19].

Receive prefixes $x(y)$ and restrictions νa *bind* the names y and a. A name that is not bound in a process is *free*. We assume free and bound names to be disjoint, and that every name is bound at most once. We use $\mathcal{F}$ to denote the names that can occur free in processes, and $\mathcal{F}(P)$ for the set of free names in P. Similarly, $\mathcal{R}$ is the set of names that can occur in restrictions, and $\mathcal{R}(P)$ the restricted names in P. In a defining equation $K(\tilde{x}) := P$, we require $\mathcal{F}(P) \subseteq \tilde{x}$ to avoid the invention of free names in a recursive call.

To ease the definition of the π-calculus semantics, we define a *structural congruence relation* $\equiv \subseteq \mathcal{P} \times \mathcal{P}$. It is the smallest congruence that allows for α-conversion of bound names, requires choice $+$ and parallel $\mid$ to be associative and commutative with $\mathbf{0}$ as neutral element, and where restrictions satisfy

$$\nu a.\mathbf{0} \equiv \mathbf{0} \quad \nu a.\nu b.P \equiv \nu b.\nu a.P \quad \nu a.(P \mid Q) \equiv P \mid \nu a.Q \text{ if } a \notin \mathcal{F}(P).$$

We also use a quasi-ordering $\preceq \subseteq \mathcal{P} \times \mathcal{P}$ among processes called *embedding*. It is the smallest relation that satisfies $\nu\tilde{a}.Q \preceq \nu\tilde{a}.(Q \mid R)$ and that is closed under structural congruence: $Q \equiv Q' \preceq R' \equiv R$ entails $Q \preceq R$. It can be shown that $\preceq$ is indeed reflexive and transitive, and thus a quasi-ordering [12]. For a set of processes $\mathcal{P}' \subseteq \mathcal{P}$, we define $\mathcal{P}'{\downarrow} := \{Q \mid \exists P \in \mathcal{P}' : Q \preceq P\}$.

The behavior of processes is given by the *reaction relation* $\rightarrow \subseteq \mathcal{P} \times \mathcal{P}$. It is the smallest relation that satisfies

$$x(z).P + M \mid \overline{x}\langle y\rangle.Q + N \rightarrow P\{y/z\} \mid Q \qquad K\lfloor\tilde{a}\rfloor \rightarrow P\{\tilde{a}/\tilde{x}\} \text{ with } K(\tilde{x}) := P,$$

and that is closed under parallel composition, restriction, as well as structural congruence. A process Q is reachable from P if $P \rightarrow^* Q$, where $\rightarrow^*$ is the reflexive transitive closure of $\rightarrow$. The *reachability set* of P, written $\mathit{Reach}(P)$, is the set of all processes reachable from P. The *transition system* of a process is the quotient of the reachable processes along structural congruence: $\mathcal{T}(P) := (\mathit{Reach}(P)/_{\equiv}, \hookrightarrow, \underline{P})$, where $\underline{P} \hookrightarrow \underline{Q}$ iff $P \rightarrow Q$.

We shall use Milner's *standard form* of processes [16], which maximizes the scope of restrictions. The formal definition is inductive. A parallel composition of sequential processes S is in standard form. If P_{sf} is in standard form then also $\nu a.P_{sf}$ is, provided $a \in \mathcal{F}(P_{sf})$.

Name Boundedness. Process $P \in \mathcal{P}$ is *name-bounded* if there is a bound on the number of restricted names that holds for all reachable processes $Q \in \mathit{Reach}(P)$. However, restricted names that do not occur free can be removed by structural congruence: $\nu a.P \equiv P$ provided $a \notin \mathcal{F}(P)$. This motivates the definition of the number of *active restrictions*: $\mathit{arn}(S) := 0$, $\mathit{arn}(P \mid Q) := \mathit{arn}(P) + \mathit{arn}(Q)$, and $\mathit{arn}(\nu a.P) := 1 + \mathit{arn}(P)$ if $a \in \mathcal{F}(P)$ and $\mathit{arn}(\nu a.P) := \mathit{arn}(P)$ otherwise.

Definition 1. *Process $P \in \mathcal{P}$ is* b-name-bounded *with $b \in \mathbb{N}$ if for all processes $Q \in \mathit{Reach}(P)$ we have $\mathit{arn}(Q) \leq b$. Process P is* name-bounded *if it is b-name-bounded for some $b \in \mathbb{N}$.*

For example, process $\nu a.K_1\lfloor a\rfloor$ with $K_1(x) := K_1\lfloor x\rfloor^2$ is 1-name-bounded, whereas $\nu b.K_2\lfloor b\rfloor$ with $K_2(y) := \nu b.(K_2\lfloor b\rfloor^2)$ is not name-bounded.

The definition of name boundedness refers to all reachable processes. This motivates the following decision problems. The *name boundedness problem (***NB***)* asks, given $P \in \mathcal{P}$, is P name-bounded? The *restricted name boundedness problem (***RNB***)* asks, given $P \in \mathcal{P}$ and $b \in \mathbb{N}$, if P is b-name-bounded.

Theorem 1. *(1)* **NB** *is r.e.-complete and (2)* **RNB** *is decidable.*

Decidability of **RNB**, from which the recursive enumerability of **NB** follows, is not obvious, as 0-name-bounded systems can already simulate Petri nets. We prove the result using an algorithm for effectively constructing coverability sets for name-bounded processes that we describe next.

3 Coverability and the Karp-Miller Construction

A process Q is said to be *coverable* from P if there exists some $R \in \mathit{Reach}(P)$ such that $Q \preceq R$. The *coverability set* of P is the set of all processes coverable from P. Equivalently, the coverability set is the downward closure, with respect to the embedding order, of the reachability set: $\mathit{Reach}(P)\downarrow$. Given a name-bounded process P, our next goal is to compute a *finite* representation for $\mathit{Reach}(P)\downarrow$.

We construct the coverability set by unfolding the reachability tree of P, and accelerating reaction sequences that can be repeated. The acceleration procedure takes a sequence $Q_1 \to^* Q_2$ and constructs a closed-form representation of all processes that are coverable with any number of iterations of the sequence. We call the procedure a Karp-Miller tree, since it closely resembles the data structure used for the coverability analysis of Petri nets [9,7].

3.1 Identity-aware Processes

In the Karp-Miller tree construction, restricted names have to be handled with care. To see the problem, reconsider $K_1(x) := K_1\lfloor x \rfloor^2$ and $K_2(y) := \nu b.(K_2\lfloor b \rfloor^2)$ from above. Then $\nu a.K_1\lfloor a \rfloor$ is name-bounded, but $\nu b.K_2\lfloor b \rfloor$ is not. In their reachability trees, reaction $\nu a.K_1\lfloor a \rfloor \to \nu a.(K_1\lfloor a \rfloor^2)$ uses copies of the same name a, while $\nu b.K_2\lfloor b \rfloor \to \nu b.(K_2\lfloor b \rfloor^2)$ forgets name b and re-creates it again.

The examples suggest that we have to track the identities of names over transitions. Inspired by [14], we introduce a notion of *identity-aware processes*. They replace restricted names νa by free names of the form (a, i) taken from a set of *instances*. Since free names are not subject to α-conversion, instances are preserved by transitions. To mimic name boundedness, transitions among identity-aware processes do not choose instances (a, i) arbitrarily, but compute the least index i that is not present in the target process. Moreover, identity-aware transitions are labelled by the newly generated instances, which allows us to distinguish them from old ones. In the example above, we obtain

$$K_1\lfloor (a,0) \rfloor \xrightarrow{\emptyset}_{ia} K_1\lfloor (a,0) \rfloor^2 \quad \text{as opposed to} \quad K_2\lfloor (b,0) \rfloor \xrightarrow{\{(b,0)\}}_{ia} K_2\lfloor (b,0) \rfloor^2.$$

Formally, an *instance of a restricted name* $a \in \mathcal{R}$ is a pair (a, i) from the set of instances $\mathcal{I} := \mathcal{R} \times \mathbb{N}$. A process is called *identity-aware* if it has the form

$$P_{ia} = S_1 \mid \ldots \mid S_n \quad \text{with} \quad \mathcal{F}(P_{ia}) \subseteq \mathcal{F} \cup \mathcal{I}.$$

As it is a parallel composition of choices and calls, there are no active restrictions. Moreover, and different from ordinary processes, P_{ia} is allowed to have some

instances *free*. We use $\mathcal{P}_{ia}$ to refer to the set of *all identitiy-aware processes*. We let $\mathcal{I}(P_{ia}) := \mathcal{F}(P_{ia}) \cap \mathcal{I}$ return the instances in process P_{ia}.

We now define a transition relation among identity-aware processes. The idea is to compute instances that represent restricted names, rather than choosing them non-deterministically. For each restriction $a \in \mathcal{R}$, we introduce the function

$$min_a(P_{ia}) := (a, k) \quad \text{where} \quad k = min\,\{i \in \mathbb{N} \mid (a, i) \notin \mathcal{I}(P_{ia})\}\,.$$

It determines the *least instance that is not free in* P_{ia}. With this function, we can turn processes in standard form P_{sf} into identity-aware processes:

$$ia(S_1 \mid \ldots \mid S_n) := S_1 \mid \ldots \mid S_n \qquad ia(\nu a.P_{sf}) := ia(P_{sf})\{min_a(ia(P_{sf}))/a\}.$$

The *identity-aware transition relation* $\rightarrow_{ia} \subseteq \mathcal{P}_{ia} \times 2^{\mathcal{I}} \times \mathcal{P}_{ia}$ is now defined by

$$P_{ia} \xrightarrow{\mathcal{FI}(P_{ia},Q_{ia})}_{ia} Q_{ia} \qquad \text{iff} \quad P_{ia} \rightarrow Q_{sf} \quad \text{and} \quad Q_{ia} = ia(Q_{sf}).$$

Here, $\mathcal{FI}(P_{ia}, Q_{ia})$ is the set of fresh instances that are determined by $ia(Q_{sf})$. We usually write $P_{ia} \rightarrow_{ia} Q_{ia}$ and only mention $\mathcal{FI}(P_{ia}, Q_{ia})$ where it is needed. When we consider transition sequences, we form the union of the instances: $P_{ia} \rightarrow_{ia} Q_{ia} \rightarrow_{ia} R_{ia}$ yields $\mathcal{FI}(P_{ia}, R_{ia}) := \mathcal{FI}(P_{ia}, Q_{ia}) \cup \mathcal{FI}(Q_{ia}, R_{ia})$. We denote the set of all identity-aware processes that are reachable from P_{ia} via the identity-aware transition relation by $Reach_{ia}(P_{ia})$. The corresponding *identity-aware transition system* is $\mathcal{T}_{ia}(P_{ia}) := (Reach_{ia}(P_{ia})/_{\equiv}, \hookrightarrow_{ia}, \underline{P_{ia}})$, where $\underline{Q_{ia}} \hookrightarrow_{ia} \underline{R_{ia}}$ iff $Q_{ia} \rightarrow_{ia} R_{ia}$. This is indeed well-defined. The identity-aware transition system is bisimilar to the original one.

Proposition 1. $\mathcal{T}(P_{sf}) \approx \mathcal{T}_{ia}(ia(P_{sf}))$.

The bisimulation that relates the transition systems of P_{sf} and $P_{ia} = ia(P_{sf})$ is $\mathcal{B} \subseteq Reach_{ia}(P_{ia})/_{\equiv} \times Reach(P_{sf})/_{\equiv}$ defined by $\underline{Q_{ia}} \;\; \mathcal{B} \;\; \underline{\nu\, \mathcal{I}(Q_{ia}).Q_{ia}}$. We call an identity-aware process P_{ia} name-bounded if $\underline{\nu\mathcal{I}(P_{ia}).P_{ia}}$ is. Equivalently, there is a finite set of instances that are used in any reachable process.

We elaborate on the shape of processes $Q_{ia} \in Reach_{ia}(P_{ia})$, making use of derivatives as introduced in [13]. Intuitively, process Q_{ia} consists of subterms of P_{ia} to which substitutions are applied. The idea of subterms is formalized by the notion of *derivatives* $\mathcal{D}(P)$. Function $\mathcal{D} : \mathcal{P} \rightarrow 2^{\mathcal{P}}$ returns the set of processes that can be found by removing prefixes, restrictions, and splitting up parallel compositions — in P and in its defining equations:

$$\begin{aligned}
\mathcal{D}(\mathbf{0}) &:= \emptyset & \mathcal{D}(K\lfloor\tilde{a}\rfloor) &:= \{K\lfloor\tilde{a}\rfloor\} \cup \mathcal{D}(Q) \text{ if } K(\tilde{x}) := Q \\
\mathcal{D}(\pi.P) &:= \{\pi.P\} \cup \mathcal{D}(P) & \mathcal{D}(M + N) &:= \{M + N\} \cup \mathcal{D}(M) \cup \mathcal{D}(N) \\
\mathcal{D}(P \mid Q) &:= \mathcal{D}(P) \cup \mathcal{D}(Q) & \mathcal{D}(\nu a.P) &:= \mathcal{D}(P).
\end{aligned}$$

Since processes rely on finitely many defining equations, the set remains finite. Derivatives do not keep track of what names are instantiated, nor what names are received in input prefixes. To correctly represent Q_{ia}, we map the names

Algorithm 1. Karp & Miller Tree Construction

procedure KM(P_{ia})
  $V := \{\mathsf{root} : P_{ia}\}$; $\rightarrow_{KM} := \emptyset$; $Work := \mathsf{root} : P_{ia}$;
  while *Work* not empty **do**
    Pop $\mathsf{n}_1 : L_1$ from *Work*;
    for all $L_1 \rightarrow_{ia} L_2$ up to $\equiv$ **do**
      if there is $\mathsf{n} : L \rightarrow^*_{KM} \mathsf{n}_1 : L_1$ such that $L_2 \equiv L \mid L_{rem}$ and
                $\mathcal{I}(L_{rem}) \cap \mathcal{FI}(L, L_2) = \emptyset$ **then**
        $L_2 := L \mid L^{\omega}_{rem}$;
      let n_2 be a new node
      $V := V \cup \{\mathsf{n}_2 : L_2\}$; $\rightarrow_{KM} := \rightarrow_{KM} \cup \{\mathsf{n}_1 : L_1 \xrightarrow{\mathcal{FI}(L_1, L_2)}_{KM} \mathsf{n}_2 : L_2\}$;
      $Work := Work \cdot (\mathsf{n}_2 : L_2)$ provided L_2 does not occur from root to n_1;
  return $(V, \rightarrow_{KM}, \mathsf{root} : P_{ia})$.
end procedure

occurring in derivatives to either free names in P_{ia} or to instances of restricted names. The corresponding set of substitutions is

$$\Sigma(P_{ia}) := \mathcal{F}(\mathcal{D}(P_{ia})) \rightarrow \mathcal{F}(P_{ia}) \cup (\mathcal{R}(P_{ia}) \times \mathbb{N}).$$

Lemma 1. *For every $Q_{ia} \in Reach_{ia}(P_{ia})$ there are $D_1, \ldots, D_n \in \mathcal{D}(P_{ia})$ and $\sigma_1, \ldots, \sigma_n \in \Sigma(P_{ia})$ so that $Q_{ia} \equiv D_1\sigma_1 \mid \ldots \mid D_n\sigma_n$.*

Note that a name-bounded P_{ia} only uses a finite number of instances from $\mathcal{R}(P_{ia}) \times \mathbb{N}$, and therefore σ_1 to σ_n are taken from a finite subset of $\Sigma(P_{ia})$.

3.2 Karp and Miller Trees

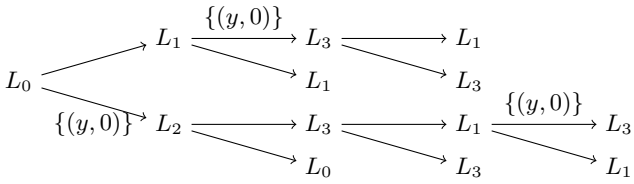
The Karp-Miller tree for a process P_{ia} is a rooted, directed tree. The nodes of the tree are labeled with either a single process reachable from P_{ia}, or a *limit process*, representing a set of processes summarizing the effect of repeating a reaction sequence. The root is labeled with P_{ia}.

The definition of limits is inspired by replication in π-calculus, and similar to [20]. A limit is either a sequential process S, a process of the form L^{ω}, or a parallel composition $L_1 \mid L_2$ of limit processes. Intuitively, limit S^{ω} represents an unbounded set of processes S^j for arbitrarily large j. We extend structural congruence to limit processes with the following rules:

$$S^{\omega} \mid S \equiv S^{\omega} \qquad S^{\omega} \mid S^{\omega} \equiv S^{\omega} \qquad (S^{\omega})^{\omega} \equiv S^{\omega} \qquad (L_1 \mid L_2)^{\omega} \equiv L_1^{\omega} \mid L_2^{\omega}$$

While decidability of structural congruence for processes with replication is problematic, it is not an issue here since ω distributes over parallel composition.

By associativity and commutativity of parallel composition and the above laws for limit processes, we can bring each limit process L into the standard form $L \equiv S_1^{k_1} \mid \ldots \mid S_n^{k_n}$, where $S_i \not\equiv S_j$ for $i \neq j$ and $k_i \in \mathbb{N} \cup \{\omega\}$. Thus, we order the sequential processes into groups of structurally congruent ones



$$L_0 := K_1\lfloor(a,0)\rfloor \mid K_3 \qquad L_2 := K_1\lfloor(a,0)\rfloor \mid \overline{(y,0)} \mid (y,0).K_3$$
$$L_1 := K_1\lfloor(a,0)\rfloor^\omega \mid K_3 \qquad L_3 := K_1\lfloor(a,0)\rfloor^\omega \mid \overline{(y,0)} \mid (y,0).K_3.$$

Fig. 1. Algorithm 1 on $K_1\lfloor(a,0)\rfloor \mid K_3$ with $K_1(x) := K_1\lfloor x\rfloor^2$ and $K_3 := \nu y.(\overline{y} \mid y.K_3)$

and join them if we find ω. With this normal form, we can understand limit L as a multiset $L : \mathcal{S} \to \mathbb{N} \cup \{\omega\}$ that assigns $L(S_i) := k_i$. If process $S \in \mathcal{S}$ does not occur in the parallel composition above, it is assigned zero. We use $Fin(L) := \{S \in \mathcal{S} \mid 1 \leq L(S) \in \mathbb{N}\}$ for the sequential processes that occur finite in L, and $Inf(L) := \{S \in \mathcal{S} \mid L(S) = \omega\}$ for those that are ω. We also extend $\preceq$ to limits in the natural way, and note that it coincides with multiset inclusion.

Algorithm 1 shows a worklist algorithm to construct the Karp-Miller tree for a process P_{ia}. The construction starts with a root node labeled with P_{ia} and unrolls the reachability tree by executing enabled reactions. This means the edges $\mathsf{n}_1 : L_1 \to_{KM} \mathsf{n}_2 : L_2$ of the Karp-Miller tree mimic identity-aware reactions, and so are labeled with sets of fresh instances. Additionally, reaction sequences leading to repeating limits $L \preceq L \mid L_{rem}$ are accelerated, provided L_{rem} does not remember newly generated instances. We define $KM(P_{ia}) := \{L \mid \text{there is some node labeled with } L \text{ in the tree}\}$. Figure 1 gives the execution of the algorithm on an example process.

To understand the subtle acceleration condition in the if-statement, consider process $K_2\lfloor(b,0)\rfloor$ with $K_2(y) := \nu b.(K_2\lfloor b\rfloor^2)$:

$$K_2\lfloor(b,0)\rfloor \xrightarrow{\{(b,0)\}}_{ia} K_2\lfloor(b,0)\rfloor^2 \xrightarrow{\{(b,1)\}}_{ia} K_2\lfloor(b,1)\rfloor^2 \mid K_2\lfloor(b,0)\rfloor.$$

After the first transition, we find $K_2\lfloor(b,0)\rfloor \preceq K_2\lfloor(b,0)\rfloor^2$. But the intersection $\mathcal{I}(K_2\lfloor(b,0)\rfloor) \cap \{(b,0)\} \neq \emptyset$ forbids us to accelerate $K_2\lfloor(b,0)\rfloor^2$ to $K_2\lfloor(b,0)\rfloor^\omega$. That this is indeed correct can be seen in the third process. It is not true that arbitrarily many processes will get to know $(b,0)$.

The following lemmas encode the correctness of the construction.

Lemma 2 (Completeness). *For every $Q_{ia} \in Reach_{ia}(P_{ia})$ there is a limit $L \in KM(P_{ia})$ and an injective substitution $\sigma : \mathcal{I}(Q_{ia}) \to \mathcal{I}(L)$ so that $Q_{ia}\sigma \preceq L$.*

Lemma 3 (Soundness). *For every $L \in KM(P_{ia})$ and every $k \in \mathbb{N}$ there is $Q_{ia} \in Reach_{ia}(P_{ia})$ with $Q_{ia}(S) = L(S)$ for all $S \in Fin(L)$, $Q_{ia}(S) \geq k$ for all $S \in Inf(L)$, and $Q_{ia}(S) = 0$ otherwise.*

Note that the previous two lemmas do not assume P_{ia} to be name-bounded. This is important in proving Theorem 1(2) below.

Lemma 4 (Termination). *Algorithm 1 terminates with input $P_{ia} \in \mathcal{P}_{ia}$ if and only if P_{ia} is name-bounded.*

If the input process is not name-bounded, completeness in Lemma 2 implies that Algorithm 1 cannot terminate. The converse direction comes with two problems.

First, we have to guarantee that we find repeating limits $\mathsf{n}_1 : L_1 \to^*_{KM} \mathsf{n}_2 : L_2 \to_{ia} L$ with $L \equiv L_1 \mid L_{rem}$. The trick is to understand limits as multisets over a finite set, and then resort to the theory of well-quasi orderings. Lemma 1 shows that we can restrict ourselves to derivatives plus substitutions: $D\sigma$ and $(D\sigma)^\omega$. This means limits of P_{ia} are multisets over the finite set $\mathcal{D}(P_{ia}) \times \Sigma(P_{ia}) \times \{\varepsilon, \omega\}$. With this finiteness, multiset inclusion and hence $\preceq$ is a well-quasi ordering by Dickson's lemma. This guarantees repetitions.

Second, acceleration has a side condition which additionally requires $\mathcal{I}(L_{rem}) \cap \mathcal{FI}(L_1, L) = \emptyset$: the processes to be accelerated should not contain instances that were generated along the way. The following lemma shows that a failure of the side condition would contradict name boundedness.

Lemma 5. *Consider Algorithm 1 with $P_{ia} \in \mathcal{P}_{ia}$ as input. If the execution encounters $\mathsf{n}_1 : L_1 \to^*_{KM} \mathsf{n}_2 : L_2 \to_{ia} L$ with $L \equiv L_1 \mid L_{rem}$ and such that $\mathcal{I}(L_{rem}) \cap \mathcal{FI}(L_1, L) \neq \emptyset$, then P_{ia} is not name-bounded.*

Note that the lemma correctly predicts name unboundedness in the example $K_2\lfloor(b, 0)\rfloor$ above. Applied in contraposition, the lemma shows that for name-bounded processes the side condition always holds: if we find repeating elements L_1 and $L \equiv L_1 \mid L_{rem}$, then we are already sure that $\mathcal{I}(L_{rem}) \cap \mathcal{FI}(L_1, L) = \emptyset$.

Theorem 2. *Consider a name-bounded process $P \in \mathcal{P}$ with $P \equiv \nu\mathcal{I}(P_{ia}).P_{ia}$. Then $Reach(P)\downarrow = \{\nu\mathcal{I}(L).L \mid L \in KM(P_{ia})\}\downarrow$ and the latter set is finite.*

Proof of Theorem 1(2). The Karp and Miller tree allows us to prove Theorem 1. To check whether $P \in \mathcal{P}$ is b-name-bounded, we execute Algorithm 1 until it either terminates or we find a limit with more than b instances. In the former case, we report that the process is b-name-bounded, in the latter that it is not.

Assume P is b-name-bounded. Then Algorithm 1 terminates by Lemma 4. By Lemma 3, it only constructs limits with up to b instances. Therefore, the second termination condition will not apply and we report P to be b-name-bounded. If P is not b-name-bounded, there is a reachable process with more than b instances. By completeness, Algorithm 1 will construct a limit with more than b instances. Then, we report P is not b-name-bounded. □

4 From Name-bounded Processes to Petri Nets

4.1 Process Bounds

To give a reduction to Petri nets, we require a notion of "pumping" instances. We introduce *process bounds* as a pumping mechanism.

Consider a sequence $\rho = P_{ia} \to_{ia} \ldots \to_{ia} Q_{ia}$ and some $p \in \mathbb{N}$. We use $\mathcal{I}_{>p}(\rho)$ to denote the *instances that were known to more than p sequential processes at some moment during* ρ. As instances can be reused, $\mathcal{I}_{>p}(\rho)$ is a multiset. It is divided into $\mathcal{I}_{>p}(\rho) = \mathcal{I}^{active}_{>p}(\rho) + \mathcal{I}^{dead}_{>p}(\rho)$. The instances that once exceeded the bound and are still active in the final process Q_{ia} are given by $\mathcal{I}^{active}_{>p}(\rho)$. Note that $\mathcal{I}^{active}_{>p}(\rho)$ is indeed a set, not just a multiset. The instances that have been forgotten along the way is the multiset $\mathcal{I}^{dead}_{>p}(\rho)$.

The definitions are by induction on the length of the sequence, where the induction step is as follows:

$$\mathcal{I}^{dead}_{>p}(\rho.Q^1_{ia} \to_{ia} Q^2_{ia}) := \mathcal{I}^{dead}_{>p}(\rho) + \mathcal{I}_{dead}$$
$$\mathcal{I}^{active}_{>p}(\rho.Q^1_{ia} \to_{ia} Q^2_{ia}) := \mathcal{I}^{active}_{>p}(\rho) \setminus \mathcal{I}_{dead} \cup \mathcal{I}_{act}.$$

The auxiliary set $\mathcal{I}_{dead}$ contains the instances that once exceeded the bound and were active up to Q^1_{ia}, but that are forgotten in the transition $Q^1_{ia} \to_{ia} Q^2_{ia}$. The set $\mathcal{I}_{act}$ contains the instances that occur in more than $p \in \mathbb{N}$ sequential processes of Q^2_{ia}. We use $|Q^2_{ia}|_{(a,i)}$ to denote the number of sequential processes in Q^2_{ia} that have (a, i) as instance:

$$\mathcal{I}_{dead} := \left\{(a,i) \in \mathcal{I}^{active}_{>p}(\rho) \mid (a,i) \notin \mathcal{I}(Q^2_{ia}) \text{ or } (a,i) \in \mathcal{FI}(Q^1_{ia}, Q^2_{ia})\right\}$$
$$\mathcal{I}_{act} := \left\{(a,i) \in \mathcal{I}(Q^2_{ia}) \mid |Q^2_{ia}|_{(a,i)} > p\right\}.$$

Choose p to be the largest number of sequential processes that know an instance (a, i) in a limit of the Karp-Miller tree:

$$p := max\left\{\ |L|_{(a,i)}\ \mid L \in KM(P_{ia}) \text{ and } (a,i) \notin \mathcal{I}(Inf(L))\right\}.$$

This p, called the *process bound*, has an interesting property. Consider a reaction sequence ρ from P_{ia} to Q_{ia}. If we replay it on the Karp-Miller tree, then an instance $(a, k) \in \mathcal{I}_{>p}(\rho)$ that is distributed to more than p sequential processes leads to S^ω. Since accelerated processes are never removed, the limit L that dominates Q_{ia} will contain S^ω. Thus, the limit not only contains the instances $\mathcal{I}(Q_{ia})$ but also the instances from $\mathcal{I}^{dead}_{>p}(\rho)$. Since there are reachable processes R_{ia} that correspond to L, we get a lower bound on the name bound of P_{ia}.

Lemma 6. *Let $P_{ia} \in \mathcal{P}_{ia}$ be b-name-bounded and let p be its process bound. For every ρ from P_{ia} to Q_{ia} we have $|\mathcal{I}^{dead}_{>p}(\rho)| + |\mathcal{I}(Q_{ia})| \leq b$.*

Phrased differently, instances which exceeded process bound p do not need to be reused by the identity-aware semantics to meet name bound b.

4.2 Petri Net Construction

Given a b-name-bounded process $P_{ia} \in \mathcal{P}_{ia}$ that has $p \in \mathbb{N}$ as a process bound, we construct a Petri net $\mathbf{N}(P_{ia}, b, p)$ that simulates the identity-aware semantics of P_{ia} in a strong sense: there is a bisimulation between the transition systems of $\mathbf{N}(P_{ia}, b, p)$ and P_{ia}. We use the standard notion of place-transition Petri nets of the form (S, T, W, M_0) with the standard firing semantics.

We describe the construction of $\mathbf{N}(P_{ia}, b, p)$ as a composition of two parts:

$$\mathbf{N}(P_{ia}, b, p) := \mathbf{Syn}(\mathbf{Ctrl}(P_{ia}, b)) \times \mathbf{Ref}(P_{ia}, b, p).$$

Petri net $\mathbf{N}(P_{ia}, b, p)$ maintains a finite set of instances $\{(a, 0), \ldots, (a, b-1)\}$ to represent each restriction $a \in \mathcal{R}(P_{ia})$. Petri net $\mathbf{Ref}(P_{ia}, b, p)$ implements reference counters for these instances in order to correctly allocate them when restrictions are encountered. The control flow of P_{ia} is captured by $\mathbf{Ctrl}(P_{ia}, b)$. While this net only models the consumption of prefixes, the synchronisation operation $\mathbf{Syn}(\cdot)$ joins send and receive transitions with complementary labels. Finally, the $\times$ operator co-ordinates the distribution and generation of instances between $\mathbf{Syn}(\mathbf{Ctrl}(P_{ia}, b))$ and $\mathbf{Ref}(P_{ia}, b, p)$. We turn to the details.

Petri net $\mathbf{Ctrl}(P_{ia}, b)$ maintains a place for each possible sequential process. These places count the number of occurrences of the corresponding process. Since there are finitely many restrictions, each with finitely many instances, as well as finitely many derivatives, the number of places in $\mathbf{Ctrl}(P_{ia}, b)$ is finite.

The transitions are derived from the places. Consider a receive

$$S \equiv x(y).\nu\tilde{a}.Q_{ia} + \ldots \quad \text{with} \quad \nu\tilde{a} = \nu a_1 \ldots \nu a_m \quad \text{and} \quad Q_{ia} \equiv S_1^{o_1} \mid \ldots \mid S_n^{o_n}.$$

The receive operation is implemented by a set of transitions, one transition t for each z in $\mathcal{F}(P_{ia}) \cup (\mathcal{R}(P_{ia}) \times [0, b-1])$ that we can receive for y and each m-tuple of instances (a_1, k_1) to (a_m, k_m) that we may use to represent a_1 to a_m. The post set of t is the set of sequential processes in $Q_{ia}\sigma$. In the above case, this is $S_1\sigma$ to $S_n\sigma$. Substitution σ maps y to z and a_i to (a_i, k_i) for $i \in [1, m]$. The weight of the arc from t to $S_i\sigma$ is given by $Q_{ia}\sigma(S_i\sigma) = o_i$, the number of occurrences of $S_i\sigma$ in $Q_{ia}\sigma$. Additionally, the transition is labeled with information used to synchronize with the reference counters.

The synchronisation operation $\mathbf{Syn}(\mathbf{Ctrl}(P_{ia}, b))$ joins transitions with complementary send and receive labels. Finally, the $\times$ connects the synchronized control flow net with the reference counters.

To limit the number of instances to $b \in \mathbb{N}$, we reinstantiate an instance (a, k) that was present in Q_{ia}^1 in a later process Q_{ia}^2, provided it has been forgotten on the path $Q_{ia}^1 \to_{ia}^+ Q_{ia}^2$. The Petri net $\mathbf{Ref}(P_{ia}, b, p)$ maintains reference counters that track the number of sequential processes S with $(a, k) \in \mathcal{I}(S)$. An instance (a, k) can be reinstantiated in case the reference counter is zero.

To check that a reference counter is zero, we use the process bound p. (Recall that Petri nets cannot implement zero-tests.) Consider a sequence ρ from P_{ia} to Q_{ia}. An instance (a, k) in ρ that, at some moment, is distributed to more than p sequential processes and that has been forgotten in Q_{ia} is a member of $\mathcal{I}_{>p}^{dead}(\rho)$. Lemma 6 ensures this instance need not be reinstantiated. The name bound is high enough so that there is another instance that is currently not in use and that can represent the restriction. As a result, once a reference count goes beyond p, the name is never recycled. The net $\mathbf{Ref}(P_{ia}, b, p)$ has three places for each instance (a, k): a place $(a, k)^\omega$ denoting this name cannot be recycled, a place (a, k) keeping the current reference count (provided the reference count has never exceeded p), and a complementary place $\overline{(a, k)}$ that ensures the sum of tokens

in (a,k) and $\overline{(a,k)}$ together is p while the name is bounded. We implement the zero-test by checking $\overline{(a,k)}$ has p tokens.

Now, a step of the process is simulated by a sequence of steps of the Petri net that update the tokens in $\mathbf{Ctrl}\,(P_{ia}, b)$, but at the same time, instantiate required names and update the reference counts.

4.3 Bisimilarity

To obtain a clean bisimulation between a name-bounded process and its Petri net semantics, we use an idea from [15]: we define distinguished *stable* markings that will actually correspond to processes, as opposed to intermediary markings that occur when we amend reference counters and allocate instances.

Consider $\mathbf{N}\,(P_{ia}, b, p) = (S, T, W, M_0)$. The set of places in $\mathbf{Syn}(\mathbf{Ctrl}\,(P_{ia}, b))$ is the disjoint union $S_{proc} \uplus S_{inter} \subseteq S$ where S_{proc} contains the process places of $\mathbf{Ctrl}\,(P_{ia}, b)$ while S_{inter} contains intermediary places that we added to allocate instances and update reference counters. Now, a marking $M \in \mathbb{N}^S$ is called *stable* if $M(s) = 0$ for all $s \in S_{inter}$. We use $\mathcal{R}_{stbl}(\mathbf{N}\,(P_{ia}, b, p))$ to denote the set of stable markings that are reachable in $\mathbf{N}\,(P_{ia}, b, p)$.

We are interested in transition sequences of $\mathbf{N}\,(P_{ia}, b, p)$ that correspond to *one* communication or identifier-call, rather than interleavings of reactions. Formally, a transition sequence $M^1 \xrightarrow{t_1 \dots t_n} M^2$ between stable markings is called *race-free* if there is a single transition t of $\mathbf{Syn}(\mathbf{Ctrl}\,(P_{ia}, b))$ that is unfolded into $t_1 \dots t_n$ in the composition $\mathbf{N}\,(P_{ia}, b, p)$. We write $M^1 \Rightarrow M^2$ if there is a race-free transition sequence between the two markings. With this, the *stable transition system* is $\mathcal{T}_{stbl}(\mathbf{N}\,(P_{ia}, b, p)) := (\mathcal{R}_{stbl}(\mathbf{N}\,(P_{ia}, b, p)), \Rightarrow, M_0)$.

To define a bisimulation relation, we have to decide which instances to mark as unbounded in the marking of $\mathbf{N}\,(P_{ia}, b, p)$. Unfortunately, a single process Q_{ia} does not carry enough information to define the related markings. We need the full transition sequence $\rho = P_{ia} \to_{ia} \dots \to_{ia} Q_{ia}$ that leads to Q_{ia}. We therefore extend the identity-aware transition system $\mathcal{T}_{ia}(P_{ia})$ to a *history-preserving identity-aware transition system* $\mathcal{T}^h_{ia}(P_{ia})$ in which states are such sequences ρ from P_{ia} to Q_{ia}. Clearly, the identity-aware transition system and its history-preserving variant are bisimilar. The history-preserving transition system carries enough information to establish a bisimulation result.

Lemma 7. $\mathcal{T}_{ia}(P_{ia}) \approx \mathcal{T}^h_{ia}(P_{ia}) \approx \mathcal{T}_{stbl}(\mathbf{N}\,(P_{ia}, b, p))$.

Assume P_{ia} has name bound $b \in \mathbb{N}$ and process bound $p \in \mathbb{N}$. Then $\mathcal{B}$ relates transition sequence ρ from P_{ia} to Q_{ia} with marking $M = M_{ctr} + M_{ref}$ if the following holds.

For the control-flow marking, we require that there is an injective substitution $\sigma : \mathcal{I}(Q_{ia}) \to \mathcal{I}(supp(M_{ctr}))$ so that $M_{ctr} \equiv Q_{ia}\sigma$.

For marking M_{ref} of the reference counter, consider $(a,i) \in \mathcal{I}^{active}_{>p}(\rho)$. We require $M_{ref}(\ [(a,i)\sigma]^\omega\ ) = 1$. For an instance $(a,i) \in \mathcal{I}(Q_{ia}) \backslash \mathcal{I}^{active}_{>p}(\rho)$, we need $M_{ref}(\ (a,i)\sigma\ ) = |Q_{ia}|_{(a,i)}$. With Lemma 6, there are $b - |\mathcal{I}(Q_{ia})| \geq |\mathcal{I}^{dead}_{>p}(\rho)|$ instances (a,k) outside the range of σ. We have $M_{ref}(\ (a,k)^\omega\ ) = 1$ for $|\mathcal{I}^{dead}_{>p}(\rho)|$ such instances. All other instances carry p tokens on the complement place.

The above constraints describe a marking that is partial in that, for each instance, the token count of only one place (a,k), $\overline{(a,k)}$, or $(a,k)^\omega$ is given. The tokens for the remaining places are uniquely determined by the following invariants. Places (a,i) and $\overline{(a,i)}$ are complements with bound p. Also places $\{(a,i), \overline{(a,i)}\}$ and $(a,i)^\omega$ are complements. This means, if $(a,i)^\omega$ carries a token, the other two are empty and vice versa. Place $(a,i)^\omega$ is safe. We can now state our second main result.

Theorem 3. *Let $P \equiv \nu\mathcal{I}(P_{ia}).P_{ia}$ be name-bounded by $b \in \mathbb{N}$ and let $p \in \mathbb{N}$ be the process bound. Then $\mathcal{T}(P) \approx \mathcal{T}_{stbl}(\mathbf{N}(P_{ia}, b, p))$.*

Proof. $\mathcal{T}(P) \approx \mathcal{T}_{ia}(P_{ia}) \approx \mathcal{T}^h_{ia}(P_{ia}) \approx \mathcal{T}_{stbl}(\mathbf{N}(P_{ia}, b, p))$, with Proposition 1 and Lemma 7. □

Corollary 1. *Reachability is decidable for name-bounded processes.*

5 Ackermann Lower Bound

We give a polynomial-time reduction from Turing machines operating on a tape of non-primitive recursive size to name-bounded processes. As a consequence, verification problems for name-bounded systems have non-primitive recursive complexity.

We give the construction of process $P(TM)$ for Turing machine TM. The behavior of $P(TM)$ is divided into three stages. In the first stage, $P(TM)$ generates up-to $A(n)$ parallel processes $W\lfloor p, Q\rfloor$ that are waiting. Here, Q is the set of states in the Turing machine that we deliberately understand as channels. Moreover, there is a single process $\nu c.\nu r.G\lfloor p, c, r, q_0\rfloor$:

$$P(TM) \to^* W\lfloor p, Q\rfloor^{A(n)} \mid \nu c.\nu r.G\lfloor p, c, r, q_0\rfloor.$$

For this generation phase, we rely on a result from Petri net theory [11]. There is a sequence of Petri nets $(N_i)_{i\in\mathbb{N}}$ where the size grows linearly and that produce up to $(A(i))_{i\in\mathbb{N}}$ tokens on a designated place. Such a Petri net N_i can be turned into a restriction-free process creating $A(i)$ copies of $W\lfloor p, Q\rfloor$.

In the second stage, process $G\lfloor p, c, r, q_0\rfloor$ aligns the waiting $W\lfloor p, Q\rfloor$ into a list of processes $C_0\lfloor l, c, r, Q\rfloor$ representing cells in the Turing tape with content 0, input channel c, and pointers l and r to the input channels of their left and right neighbor. The process $G\lfloor p, c, r, q_0\rfloor$ recursively converts $W\lfloor p, Q\rfloor$ to cells, and can non-deterministically decide that the current cell with input channel c is the last one in the list. In this case it sends the initial state q_0 of TM to the cell, which starts the simulation of the Turing machine.

The simulation of the Turing machine is the third stage in the behavior of $P(TM)$. Each process $C_i(l, c, r, Q)$, with $i \in \{0, 1\}$ as current content, waits to receive the head pointer and the current state of TM. On receiving the current state, the process executes one transition of the machine and updates its content, while sending the successor state to its left or right neighbor, based on the transition.

Theorem 4. *(1) Reachability and coverability are non-primitive recursive for name-bounded processes. (2) There is no primitive recursive translation of*

name-bounded processes into Petri nets that preserves coverability. (3) There is no primitive recursive bound on the size of name and process bounds.

Theorem 4 shows the Karp-Miller procedure is asymptotically optimal.

References

1. Amadio, R., Meyssonnier, C.: On decidability of the control reachability problem in the asynchronous π-calculus. Nord. J. Comp. 9(1), 70–101 (2002)
2. Bansal, K., Koskinen, E., Wies, T., Zufferey, D.: Structural counter abstraction. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 62–77. Springer, Heidelberg (2013)
3. Busi, N., Gorrieri, R.: Distributed semantics for the π-calculus based on Petri nets with inhibitor arcs. J. Log. Alg. Prog. 78(1), 138–162 (2009)
4. Busi, N., Zavattaro, G.: Deciding reachability problems in Turing-complete fragments of Mobile Ambients. Math. Struct. Comp. Sci. 19(6), 1223–1263 (2009)
5. Dam, M.: Model checking mobile processes. Inf. Comp. 129(1), 35–51 (1996)
6. Devillers, R., Klaudel, H., Koutny, M.: A compositional Petri net translation of general π-calculus terms. For. Asp. Comp. 20(4-5), 429–450 (2008)
7. Finkel, A., Goubault-Larrecq, J.: The theory of WSTS: The case of complete WSTS. In: Haddad, S., Pomello, L. (eds.) PETRI NETS 2012. LNCS, vol. 7347, pp. 3–31. Springer, Heidelberg (2012)
8. He, C.: The decidability of the reachability problem for CCS! In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 373–388. Springer, Heidelberg (2011)
9. Karp, R.M., Miller, R.E.: Parallel program schemata. J. Comput. Syst. Sci. 3(2), 147–195 (1969)
10. Lipton, R.J.: The reachability problem requires exponential space. Technical report, Yale University, Department of Computer Science (1976)
11. Mayr, E.W., Meyer, A.R.: The complexity of the finite containment problem for Petri nets. JACM 28(3), 561–576 (1981)
12. Meyer, R.: On boundedness in depth in the π-calculus. In: *IFIP TCS*. IFIP, vol. 273, pp. 477–489. Springer, Heidelberg (2008)
13. Meyer, R.: A theory of structural stationarity in the π-calculus. Acta Inf. 46(2), 87–137 (2009)
14. Meyer, R., Gorrieri, R.: On the relationship between π-calculus and finite place/transition Petri nets. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 463–480. Springer, Heidelberg (2009)
15. Meyer, R., Khomenko, V., Hüchting, R.: A polynomial translation of π-calculus (FCP) to safe Petri nets. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 440–455. Springer, Heidelberg (2012)
16. Milner, R.: Communicating and Mobile Systems: the π-Calculus. CUP (1999)
17. Montanari, U., Pistore, M.: Checking bisimilarity for finitary π-calculus. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 42–56. Springer, Heidelberg (1995)
18. Rackoff, C.: The covering and boundedness problems for vector addition systems. Theor. Comp. Sci. 6(2), 223–231 (1978)
19. Sangiorgi, D., Walker, D.: The π-calculus: a Theory of Mobile Processes. CUP (2001)
20. Wies, T., Zufferey, D., Henzinger, T.A.: Forward analysis of depth-bounded processes. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 94–108. Springer, Heidelberg (2010)

A Hierarchy of Expressiveness in Concurrent Interaction Nets

Andrei Dorman[1] and Damiano Mazza[2]

[1] Dipartimento di Filosofia, Università degli Studi Roma Tre
and Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS
Andrei.Dorman@lipn.univ-paris13.fr
[2] CNRS, UMR 7030, LIPN, Université Paris 13, Sorbonne Paris Cité
Damiano.Mazza@lipn.univ-paris13.fr

Abstract. We give separation results, in terms of expressiveness, concerning all the concurrent extensions of interaction nets defined so far in the literature: we prove that multirule interaction nets (of which Ehrhard and Regnier's differential interaction nets are a special case) are strictly less expressive than multiwire interaction nets (which include Beffara and Maurel's concurrent nets and Honda and Laurent's version of polarized proof nets); these, in turn, are strictly less expressive than multiport interaction nets (independently introduced by Alexiev and the second author), although in a milder way. These results are achieved by providing a notion of barbed bisimilarity for interaction nets which is general enough to adapt to all systems but is still concrete enough to allow (hopefully) convincing separation results. This is itself a contribution of the paper.

Keywords: Interaction nets, Expressiveness in concurrency, Behavioral equivalences.

1 Introduction

Interaction nets were introduced by Yves Lafont [10] as a model of distributed and deterministic computation, inspired by proof nets of multiplicative linear logic [11]. To this date, they have earned a prominent place in the theory of the optimal implementation of the λ-calculus (as *sharing graphs* [13,2]) and functional programming languages in general [15,16], as well as in the related field known as the geometry of interaction [7], for which they are the most natural syntax.

The main interest of interaction nets lies in the fact that they provide a simple yet extremely powerful paradigm for representing a vast variety of computational models, ranging from Turing machines to functional languages, passing through cellular automata and term-graphs, all this by respecting the essential idea that computation is local and the cost of elementary steps is bounded by a constant. Additionally, interaction nets provide a pleasant and intuitive graphical representation of programs, similar in style and spirit to string diagrams for monoidal categories.

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 197–211, 2013.

Let us define interaction nets in a nutshell. We start with an *alphabet*, *i.e.*, a set of *symbols*, $\alpha, \beta, \ldots$, each with a given arity. Given a denumerable set of *ports* $x, y, z, \ldots$, the atomic components of interaction nets are:

- *agents* (or *cells*), of the form $\alpha(x; \tilde{y})$, where $x, \tilde{y}$ are ports, with the length of the list $\tilde{y}$ matching the arity of α, and x being the *principal port* of the cell;
- *wires*, which are multisets of exactly two ports, written $[x, y]$.

A *net* is a multiset of agents and wires, in which every port appears *at most twice*. We write nets as

$$\alpha_1(x_1; \tilde{y}_1) \mid \cdots \mid \alpha_m(x_m; \tilde{y}_m) \mid [z_1, u_1] \mid \cdots \mid [z_n, u_n],$$

using a notation reminiscent of process calculi (especially the solos calculus [14]). A port appearing exactly once in a net μ is *free*, and $\mathrm{fp}(\mu)$ is the set of free ports of μ; all other ports are *bound*, and may be renamed as usual by α-equivalence. We also equate nets obtained by "fusing" or "absorbing" wires:

$$[x, y] \mid [y, z] \equiv [x, z], \qquad \mu \mid [y, x] \equiv \mu\{x/y\} \quad \text{if } y \in \mathrm{fp}(\mu)$$

(note that $[x, y] \mid [y, x] \equiv [x, x]$ are legitimate nets, that is why nets and wires are *multisets*, not just sets).

An interaction net system is obtained by choosing an alphabet and fixing a set of *interaction rules* of the form

$$\alpha(x; \tilde{y}) \mid \beta(x; \tilde{z}) \quad \to \quad \nu\{\tilde{y}, \tilde{z}\},$$

where $\nu\{\tilde{y}, \tilde{z}\}$ is a net whose free ports are exactly $\tilde{y}, \tilde{z}$. An essential requirement is that there is *at most one* interaction rule for every unordered pair of symbols α, β in the alphabet. For instance, if we take $\mathbf{0}$, $\mathbf{s}$ and $+$ as symbols, of respective arities 0, 1 and 2, and if we fix the rules

$$\mathbf{0}(x) \mid +(x; y, z) \to [y, z], \qquad \mathbf{s}(x; u) \mid +(x; y, z) \to +(u; v, z) \mid \mathbf{s}(y; v),$$

we obtain a simple system for unary arithmetic with sum. Indeed, if we set $\underline{n}\{x\} = \mathbf{0}(v_1) \mid \mathbf{s}(v_2; v_1) \mid \cdots \mid \mathbf{s}(x; v_n)$, we invite the reader to check that $\underline{m}\{x\} \mid \underline{n}\{z\} \mid +(x; y, z) \to^* \underline{m+n}\{y\}$ in $m + 1$ reduction steps. Although this particular example does not exhibit any parallelism (there is at most one reduction possible at each step), it is easy to imagine situations in which an arbitrary number of reductions may be fired at the same time.

We mentioned above that interaction nets have a simple and natural graphical representation, owing to their kinship with proof nets. Actually, in the existing literature nets are usually presented primarily in that way [10,15,16]. For instance, the rules for the above system for unary arithmetic would be defined graphically as in Fig. 1. In this paper, we stick to a textual representation, which has the advantage of being more concise and, we feel, more easily formalizable (although, arguably, much less visually appealing).

As a rewriting system, interaction nets are strongly confluent: this is because rewriting only acts on nets of the form $\alpha(x; \tilde{y}) \mid \beta(x; \tilde{z})$, called *active pairs*,

Fig. 1. The interaction rules for sum in unary arithmetic

and these can never overlap (*i.e.*, there are no critical pairs) because x already appears twice and hence nowhere else. Strong confluence implies that Lafont's model is strictly deterministic. However, the parallelism of interaction nets suggests that, by endowing them with some form of non-determinism, it may be possible to obtain interesting models of concurrent computation.

The first to study such non-deterministic extensions was Vladimir Alexiev [1], who immediately realized that there are essentially three independent ways of altering Lafont's definition so as to inject non-determinism in the model:[1]

multirules: relaxing the requirement that there be at most one rule for every active pair;

multiwires: up to equivalence, an active pair has the form $\alpha(x, \tilde{s}) \mid \beta(y; \tilde{t}) \mid [x, y]$; if we allow wires connecting more than two ports, we obtain nets such as $\alpha(x; \tilde{s}) \mid \beta(y; \tilde{t}) \mid \gamma(z; \tilde{u}) \mid [x, y, z]$, in which active pairs overlap;

multiports: a further alternative is allowing agents to have more than one principal port, *i.e.*, more than one port on which they may interact with other cells. We thus obtain nets such as $\alpha(x, y; \tilde{s}) \mid \beta(x; \tilde{t}) \mid \gamma(y; \tilde{u})$, in which x and y are both principal for α, so one cell belongs to two active pairs.

Alexiev studied, to some extent, the inter-encodability of the various extensions, and exhibited an encoding of the replication-free π-calculus in the multiport variant, as proof that concurrent computation becomes possible in such extension of interaction nets.

In the ensuing years, other people independently defined or used similar non-deterministic variants of interaction nets, always in connection with concurrency: Ehrhard and Regnier's differential interaction nets [5] are in fact a special case of multirule interaction nets, in which Ehrhard and Laurent proposed an encoding of the π-calculus [4]; Beffara and Maurel's concurrent nets [3] use the multiwire extension, which is also mentioned by Yoshida in her work on concurrent combinators [22] and is implicit in the formulation of polarized proof nets used by Honda and Laurent to provide a correspondence with the asynchronous π-calculus [9]; and multiport interaction nets were shown by the second author to be able to encode the full π-calculus [17], improving Alexiev's result.

This leaves us with a natural question: are all these concurrent extensions of interaction nets equally expressive? Although, as mentioned above, encodings of the π-calculus were proposed for each one of these extensions, such encodings

[1] Actually, Alexiev considered four extensions, but the fourth one has never been used in the literature.

are so different in nature and their correctness is proved using such *ad hoc* arguments that, up to date, the relative expressiveness of each concurrent variant of interaction nets with respect to the others is far from clear.

The situation is further complicated by the absence, in concurrent interaction nets, of a notion of behavioral equivalence, an essential tool for comparing concurrent calculi. This is the ultimate reason why correctness proofs must resort to somewhat contrived arguments: in [4], correctness crucially depends on the definition of a labelled transition system on differential interaction nets which is quite *ad hoc* (if not highly questionable, see [18]); in [17], an operational correspondence between π-calculus reduction and interaction nets reduction is achieved through a notion of *readback* in interaction nets, which heavily depends on the encoding. Finally, although the authors of [9] do not need to address the problem because their operational correspondence is exact (*i.e.*, it is close to an "operational isomorphism"), the π-calculus they consider is asynchronous, while the other two encodings consider the synchronous one, and after Palamidessi's work [19] we know that the difference is not anodyne.

A comparison between the various non-deterministic extensions of interaction nets is attempted in the already mentioned work of Alexiev [1]. His conclusion is that the multiwire and multiport extensions are equivalent, whereas multirules are strictly less expressive. However, we feel that Alexiev's approach is not technically satisfactory: for the positive results, the question of defining a behavioral equivalence on interaction nets is not addressed and the correctness of the encodings is left unproven;[2] and the negative result is based on a severely constrained definition of translation (the nature principal/auxiliary of free ports must be preserved), which makes it less convincing than what one would hope. Finally, Alexiev never considers divergence, which is, as we will see, a key notion to capture the difference between multiwire and multiport nets.

In light of the above discussion, our starting point will be to propose a notion of behavioral equivalence for concurrent interaction nets, which is based in turn on giving a definition of "barb" in interaction nets. Our solution is to adopt a sort of "may testing" approach: we write $\mu{\downarrow_x}$ if there exists a net o such that $\mathrm{fp}(o) \cap \mathrm{fp}(\mu) = \{x\}$ and such that $\mu \mid o$ generates an "observable" computation. Since reduction rules in interaction net systems may be virtually *anything*, it is hopeless to define once and for all which computations are observable, regardless of the specific system. So we stipulate that observability comes with the definition of interaction net system itself: there is a non-empty set of "observable" interaction rules and an observable computation is a reduction sequence containing an observable reduction step (furthermore, we must require that, in $\mu \mid o$, such a sequence truly comes from the interaction of μ and o and is not already present in μ or o alone). In other words, our barbs are parametric in a choice of observable reduction rules.

[2] At p. 64 of [1], Alexiev states "[W]e don't prove formally the faithfulness of our translations, but we introduce them gradually and give comprehensive examples, so we hope that we have made their faithfulness believable".

Once barbs are given, barbed bisimulation and barbed congruence are defined in the standard way. Then, we proceed to introduce the notion of *translation* which will be the subject of our separation results. This is based on an almost straightforward reformulation, in interaction nets, of fairly standard properties which are asked of encodings between process algebras. We take as main reference Gorla's work [8], whose thorough analysis of the literature on encoding and separation results approaches exhaustiveness. Among other papers which are a guideline to our work we mention [20,19].

In synthesis, the most important properties of our translations are the preservation of the degree of distribution, operational correspondence (completeness and correctness with respect to reductions, up to barbed congruence) and a bisimulation condition which excludes trivial encodings (such as those mapping every source net to the empty net).

Finally, our separation results technically take the the following form:

- there is a system of multiwire (or multiport) interaction net which cannot be translated into any interaction net system using only multirules (Theorem 1);
- there is a system of multiport interaction nets which cannot be translated into any interaction net system using only multirules and multiwires, *without introducing divergence* (Theorem 2).

The key to the first result is formalizing the fact that the multirule extension only provides interaction nets with "internal" non-determinism. For this, we introduce *must observability* $\mu\Downarrow_x$, which is defined by the fact that, for all μ' such that $\mu \to^* \mu'$, we have $\mu' \to^* \mu''\downarrow_x$. In other words, whatever happens inside μ, the port x will always be observable. Then, we verify that, in multirule systems, must observability may not be altered by interaction with contexts: if $\mu\Downarrow_x$ and ν does not contain x, then $(\mu \mid \nu)\Downarrow_x$. This is false in multiwire and multiport systems, and gives easily a separation argument.

The second result owes virtually everything to Palamidessi's idea for separating asynchrony from synchrony in the π-calculus [19]. Indeed, the proof is more or less a reformulation, in multiport interaction nets, of a simple leader election problem in a symmetric network, which we show to be translatable in multiwire systems only introducing divergence, because multiwires (and multirules) alone are not able to synchronously "break the symmetry".

2 Concurrent Interaction Nets

Throughout the paper, we fix a denumerably infinite set of *ports*, ranged over by lowercase Latin letters. We write $\tilde{x}$ to denote a finite sequence of ports $x_1, \ldots, x_n$ such that every port appears at most twice in the sequence; n is said to be the *length* of $\tilde{x}$. If ports appear at most once, we say that $\tilde{x}$ is *repetition-free*.

Definition 1 (Net). *An* alphabet *is a pair* $\Sigma = (|\Sigma|, \deg)$*, where* $|\Sigma|$ *is a set and* $\deg : |\Sigma| \to \mathbb{N}$ *is the* degree function*.*

A cell*, or* agent*, on the alphabet* Σ *is an expression of the form* $\alpha(\tilde{x})$*, where* $\alpha \in |\Sigma|$ *and* $\tilde{x}$ *is of length* $\deg(\alpha)$*.*

A k-connector *is a multiset of cardinality* $k \in \mathbb{N}$ *of ports, containing at most two occurrences of every port, denoted by* $[\tilde{x}]$*. A* 2*-connector is called a* wire*; a* k*-connector with* $k = 1$ *or* $k \geq 3$ *is called a* multiwire*.*

A net *on an alphabet* Σ *is a finite multiset of connectors and agents on* Σ *in which every port appears at most twice. A net is* simply-wired *if it contains no multiwire.*

The set of free ports *of a net* μ*, denoted by* $\mathrm{fp}(\mu)$*, is the set of ports appearing exactly once in* μ*. The ports appearing twice in a net are called* bound*. We identify any two nets which may be obtained one from the other by an injective renaming of their bound ports (this is* α-equivalence*).*

We denote by $\mu\{y/x\}$ *the net* μ *in which the only free occurrence of* x *is replaced by* y*. The notation is extended to sequences (i.e.,* $\mu\{\tilde{y}/\tilde{x}\}$*) with the obvious meaning.*

Definition 2 (Juxtaposition). *Given two nets* μ, ν*, we denote by* $\mu \mid \nu$ *the net obtained by renaming (using* α*-equivalence) the bound ports of* μ *and* ν *so that the two nets have no bound name in common, and by taking then the standard multiset union.*

Note that, unlike usual process calculi, the symbol | is not part of the syntax, it is an operation defined on nets. It is obviously commutative and has the empty net, denoted by 0, as neutral element. It is not associative in general; however, for $\mu \mid (\nu \mid \rho)$ and $(\mu \mid \nu) \mid \rho$ to be equal, it is enough that $\mathrm{fp}(\mu) \cap \mathrm{fp}(\nu) \cap \mathrm{fp}(\rho) = \emptyset$. More in general, if $\mu_1, \ldots, \mu_n$ are such that, for all pairwise distinct i, j, k, $\mathrm{fp}(\mu_i) \cap \mathrm{fp}(\mu_j) \cap \mathrm{fp}(\mu_k) = \emptyset$, then the expression $\mu_1 \mid \cdots \mid \mu_n$ is not ambiguous. Such a notation will always be used under this assumption in the sequel.

In the rest of the paper, by *congruence* on nets we mean an equivalence relation $\sim$ such that $\mu \sim \nu$ implies that for every net ρ, $\rho \mid \mu \sim \rho \mid \nu$.

Definition 3 (Structural congruence). Structural congruence*, denoted by* $\equiv$*, is the smallest congruence on nets satisfying the following:*

$$\begin{array}{rrcll} \text{0-connector:} & \mu \mid [\,] & \equiv & \mu & \\ \text{Fusion:} & [\tilde{x}, a] \mid [a, \tilde{y}] & \equiv & [\tilde{x}, \tilde{y}] & \\ \text{Wire:} & \mu \mid [a, x] & \equiv & \mu\{x/a\} & \text{if } a \in \mathrm{fp}(\mu) \end{array}$$

In the wire rule, we may further suppose that a appears in a cell (and not a connector) of μ, otherwise the rule is already subsumed by fusion.

It is sometimes useful to consider the "pure" structure of a net, abstracting from the specific names of its free ports. This is the reason behind the following notion.

Definition 4 (Mask). *We fix two infinite sequences of* reserved ports $(p_i)_{i \in \mathbb{N}}$ *and* $(q_i)_{i \in \mathbb{N}}$*. Any net on the alphabet* Σ *whose free ports are all reserved is called a* mask*. We suppose that no net other than a mask has reserved free ports. By* $\tilde{p}$ *and* $\tilde{q}$ *we will mean the sequences* $p_1, \ldots, p_m$ *and* $q_1, \ldots, q_n$*, resp., with* m *and* n *depending on the context.*

Quite obviously, every net μ whose free ports are in the repetition-free sequence $\tilde{x}$ may be seen as the "instantiation" of a mask μ_0, which is nothing but μ with

its free ports suitably renamed: $\mu = \mu_0\{\tilde{x}/\tilde{p}\}$. The reason why we need a second sequence of reserved ports $(q_i)_{i\in\mathbb{N}}$ will be clarified shortly.

In what follows, we denote by $\widetilde{\mathcal{M}(\Sigma)}$ the set of finite repetition-free sequences of masks on Σ. We denote by $\|\xi\|$ the length of such a sequence ξ.

Definition 5 (Interaction scheme). *An* interaction scheme *on an alphabet Σ is a function $\bowtie : |\Sigma| \times \mathbb{N} \times |\Sigma| \times \mathbb{N} \to \widetilde{\mathcal{M}(\Sigma)}$ such that:*

1. *if $\|\bowtie(\alpha, i, \beta, j)\| > 0$, then $1 \leq i \leq m = \deg\alpha$ and $1 \leq j \leq n = \deg\beta$, and $\alpha = \beta$ implies $i \neq j$;*
2. *in that case, the k-th mask in the sequence $\bowtie(\alpha, i, \beta, j)$ is denoted by $\alpha_i \overset{k}{\bowtie} \beta_j$ and, for all k, its free ports are exactly $p_1, \ldots, p_{i-1}, p_{i+1}, p_m, q_1, \ldots, q_{j-1}, q_{j+1}, \ldots, q_n$;*
3. *furthermore, for all (α, i, β, j), $\|\bowtie(\beta, j, \alpha, i)\| = \|\bowtie(\alpha, i, \beta, j)\|$ and, for all $1 \leq k \leq \|\bowtie(\alpha, i, \beta, j)\|$, $\beta_j \overset{k}{\bowtie} \alpha_i = \alpha_i \overset{k}{\bowtie} \beta_j\{\tilde{q}/\tilde{p}, \tilde{p}/\tilde{q}\}$.*

An interaction scheme defines rules to reduce active pairs. There may be several interaction rules for the same active pair, this is why $\bowtie(\alpha, i, \beta, j)$ is a list of nets, not just a net. Condition 2 says that the ports not partipating in the interaction are preserved by the rules. Condition 3 states that rules are symmetric when we swap symbols. Note that condition 1 stipulates that interaction rules are defined only between cells carrying different symbols or between different principal ports. This condition, which is present in the original definition of [10], was later relaxed by Lafont himself [12]. However, in this paper we adopt the more restrictive version, on the grounds that it is verified by all systems relevant to our work [5,3,9,17].

Definition 6 (Interaction net system). *An* interaction net system *(INS) is a triple $\mathcal{S} = (\Sigma_\mathcal{S}, \bowtie_\mathcal{S}, \mathcal{O}_\mathcal{S})$ where $\Sigma_\mathcal{S}$ is an alphabet, $\bowtie_\mathcal{S}$ is an interaction scheme on $\Sigma_\mathcal{S}$ and $\mathcal{O}_\mathcal{S} \subseteq |\Sigma_\mathcal{S}| \times \mathbb{N} \times |\Sigma_\mathcal{S}| \times \mathbb{N} \times \mathbb{N}$ is non-empty and such that $(\alpha, i, \beta, j, k) \in \mathcal{O}_\mathcal{S}$ implies that $\|\bowtie(\alpha, i, \beta, j)\| = l > 0$ and $1 \leq k \leq l$, and that $(\beta, j, \alpha, i, k) \in \mathcal{O}_\mathcal{S}$. Subscripts are omitted when clear from the context.*

The set $\mathcal{O}_\mathcal{S}$ specifies the *observable rules* of $\mathcal{S}$: $(\alpha, i, \beta, j, k) \in \mathcal{O}_\mathcal{S}$ means that the k-th rule for the interaction between port i of an α cell and port j of a β cell is observable. The meaning of observable rules will be explained in Sect. 3.

Definition 7 (Reduction). *The* reduction relation $\to_\mathcal{S}$ *of an INS $\mathcal{S}$ is defined as follows:*

$$\frac{\alpha_i \overset{k}{\bowtie} \beta_j \text{ defined}}{\alpha(\tilde{x}) \mid \beta(\tilde{y}) \mid [x_i, y_j, z] \;\to_\mathcal{S}\; \alpha_i \overset{k}{\bowtie} \beta_j\{\tilde{x}/\tilde{p}, \tilde{y}/\tilde{q}\} \mid [z]}\ \text{INTERACTION}$$

$$\frac{\mu \to_\mathcal{S} \mu'}{\mu \mid \nu \to_\mathcal{S} \mu' \mid \nu}\ \text{CONTEXT} \qquad \frac{\mu \equiv \mu' \quad \mu' \to_\mathcal{S} \nu' \quad \nu' \equiv \nu}{\mu \to_\mathcal{S} \nu}\ \text{STRUCT}$$

We denote by $\to_{\mathcal{S}}^*$ *the reflexive-transitive closure of* $\to_{\mathcal{S}}$*. A net structurally congruent to the net on the left side of the* INTERACTION *rule is called an* (α_i, β_j)-active pair*. Clearly,* $\mu \to_{\mathcal{S}} \nu$ *only if some* (α_i, β_j)*-active pair is reduced, using the k-th rule for* (α, i, β, j)*. When we need to specify it, we write* $\mu \xrightarrow{\alpha_i \beta_j}_k \nu$*.*

In an INS $\mathcal{S}$*, given* $\alpha \in |\Sigma_{\mathcal{S}}|$*, we say that the i-th port of* α *is* principal *if* $\| \bowtie_{\mathcal{S}}(\alpha, i, \beta, j)\| > 0$ *or* $\| \bowtie_{\mathcal{S}}(\beta, j, \alpha, i)\| > 0$ *for some* $\beta \in |\Sigma_{\mathcal{S}}|$*. Otherwise, it is called* auxiliary*.*

To improve readability, it is convenient to assume principal ports to be always the "leftmost" in the list of ports of a cell, and to use the notation $\alpha(x_1, \ldots, x_m; y_1, \ldots, y_n)$ for a cell whose symbol α is of degree $m + n$ and has m principal ports. If all ports are principal, the semicolon is omitted.

In practice, when defining an interaction net system it is convenient to specify the interaction scheme directly by giving rewriting rules of the form

$$\alpha(\tilde{x}) \mid \beta(\tilde{y}) \to \nu_1 + \cdots + \nu_l$$

where $\tilde{x}, \tilde{y}$ are repetition-free, $x_i = y_j = z$ for some i, j, and $\mathrm{fp}(\nu_k) = \{\tilde{x}, \tilde{y}\} \setminus \{z\}$. It is then intended that $\alpha_i \overset{k}{\bowtie} \beta_j$ is defined and equal to $\nu_k\{\tilde{p}/\tilde{x}, \tilde{q}/\tilde{y}\}$ (and this automatically defines also $\beta_j \overset{k}{\bowtie} \alpha_i$).

We conclude the section by introducing some terminology. An INS is:

- *multiport* if it has a symbol with more than one principal port; otherwise, it is *uniport*;
- *simply-wired* if all reduction rules introduce simply-wired nets (in that case, one usually restricts to simply-wired nets);

3 Barbs and Translations

In the following, we fix an arbitrary INS.

Definition 8 (Residue, interreduction). *Given an active pair* ϕ *of a net* μ *and a reduction* $\mu \to \mu'$ *reducing an active pair* ψ*, we have two possibilities: either* ϕ *and* ψ *share a cell (the extreme case being* $\phi = \psi$*), or they are disjoint. In the first case,* ϕ *has no residue in* μ'*; in the second case, the cells of* ϕ *are left untouched by the reduction, and* μ' *contains an active pair* ϕ' *which is "the same" as* ϕ*. This is its* residue *in* μ'*. The notion of residue is extended to reductions of arbitrary length in the obvious way.*

Let μ *be a net, and let* F *be a set of active pairs of* μ*. We say that a reduction* $\mu \to^* \mu'$ *is* F-legal *if it reduces no active pair of* F *nor any of their residues.*

Let μ, ν *be two nets, and let* F, G *be the set of* all *of their respective active pairs. An* interreduction *of* $\mu \mid \nu$ *is a reduction which is* $F \cup G$*-legal (juxtaposition may create active pairs not in* $F \cup G$*; this is why the definition is sensible).*

Definition 9 (Barbed bisimilarity). *Let* $\mathcal{S}$ *be an INS. We say that a reduction step* $\mu \xrightarrow{\alpha_i \beta_j}_k \nu$ *is* observable *if* $(\alpha, i, \beta, j, k) \in \mathcal{O}_{\mathcal{S}}$*.*

We write $\mu{\downarrow_x}$ *if there exists a net* o *such that* $\mathrm{fp}(\mu) \cap \mathrm{fp}(o) = \{x\}$ *and an interreduction of* $\mu \mid o$ *containing an observable step. We write* $\mu{\Downarrow_x}$ *if* $\mu \to^* \mu'{\downarrow_x}$ *and we say that* o *is an* observer *of* x *in* μ.

Let $\mathcal{S}, \mathcal{T}$ *be two INSs. A (weak)* barbed $(\mathcal{S},\mathcal{T})$-bisimulation *is a binary relation* $\mathcal{B} \subseteq \mathcal{S} \times \mathcal{T}$ *on nets s.t.* $\mathcal{B}(\mu,\nu)$ *implies*

- *for every port* x, $\mu{\downarrow_x}$ *implies* $\nu{\Downarrow_x}$ *and* $\nu{\downarrow_x}$ *implies* $\mu{\Downarrow_x}$;
- $\mu \to_{\mathcal{S}} \mu'$ *implies that there exists* ν' *s.t.* $\nu \to^*_{\mathcal{T}} \nu'$ *and* $\mathcal{B}(\mu',\nu')$;
- $\nu \to_{\mathcal{T}} \nu'$ *implies that there exists* μ' *s.t.* $\mu \to^*_{\mathcal{S}} \mu'$ *and* $\mathcal{B}(\mu',\nu')$.

If there exists a barbed $(\mathcal{S},\mathcal{T})$*-bisimulation* $\mathcal{B}$ *such that* $\mathcal{B}(\mu,\nu)$, *we say that* μ *and* ν *are barbed bisimilar and write* $\mu \;{}_{\mathcal{S}}\dot{\approx}_{\mathcal{T}}\; \nu$ *(we drop the subscripts when the context is clear).*

Barbed congruence *for* $\mathcal{S}$, *denoted by* $\simeq^{\mathrm{c}}_{\mathcal{S}}$, *is the greatest congruence contained in* ${}_{\mathcal{S}}\dot{\approx}_{\mathcal{S}}$.

The above definition of barb may be applied to standard name-passing calculi: there is only one reduction rule (i/o synchronization), which must be observable (by the definition, the set of observable rules is non-empty), and we thus obtain the usual barbs. The concept of interreduction is necessary to guarantee that the observable reduction step does not come from active pairs already present in μ or, worse, in the observer o.

In the following definition, by "net" we mean "net or mask".

Definition 10 (Translation). *Let* $\mathcal{S}, \mathcal{T}$ *be INSs. A* translation *from* $\mathcal{S}$ *to* $\mathcal{T}$ *is a map* $[\![\cdot]\!]$ *from nets of* $\mathcal{S}$ *to nets of* $\mathcal{T}$ *s.t., for all nets* μ, μ' *of* $\mathcal{S}$*:*

Homomorphism: $[\![0]\!] = 0$ *and* $[\![\mu \mid \mu']\!] = [\![\mu]\!] \mid [\![\mu']\!]$;

Port Invariance: *for every mask* μ_0 *of* $\mathcal{S}$, $\mathrm{fp}([\![\mu_0]\!]) = \mathrm{fp}(\mu_0)$, *and if* $\mu = \mu_0\{\tilde{x}/\tilde{p}\}$ *(cf. observation after Definition 4), we have* $[\![\mu]\!] = [\![\mu_0]\!]\{\tilde{x}/\tilde{p}\}$;

Operational Correspondence:
- $\mu \to_{\mathcal{S}} \mu'$ *implies* $[\![\mu]\!] \to^*_{\mathcal{T}} \simeq^{\mathrm{c}}_{\mathcal{T}} [\![\mu']\!]$;
- $[\![\mu]\!] \to^*_{\mathcal{T}} \nu$ *implies* $\exists$ *a net* μ' *of* $\mathcal{S}$ *s.t.* $\mu \to^*_{\mathcal{S}} \mu'$ *and* $\nu \to^*_{\mathcal{T}} \simeq^{\mathrm{c}}_{\mathcal{T}} [\![\mu']\!]$;

Bisimulation: $\mu \;{}_{\mathcal{S}}\dot{\approx}_{\mathcal{T}}\; [\![\mu]\!]$.

A translation does not introduce divergence *if, whenever* $[\![\mu]\!]$ *diverges,* μ *diverges.*

All three properties defining translations are more or less standard [8,20]. The homomorphism condition guarantees that the degree of distribution is preserved by translations and is common in separation results [19]. Port invariance simply states that the interface of a net is preserved by a translation, and that the translation itself does not depend on the actual names of ports. Operational correspondence is a natural property to ask of an encoding, although we will not use it. On the contrary, the bisimulation condition will be essential. It corresponds to what Gorla [8] calls "success sensitiveness", in that it excludes trivial translations which would otherwise be validated by the other three conditions (such as an encoding mapping every net with free ports $x_1, \ldots, x_n$ to the net $[x_1], \ldots, [x_n]$). Furthermore, bisimulation (with the homomorphism property) implies the adequacy and relative completeness of translations with respect to barbed congruence (of the respective systems):

adequacy: $[\![\mu]\!] \simeq^{\mathrm{c}}_{\mathcal{T}} [\![\mu']\!]$ implies $\mu \simeq^{\mathrm{c}}_{\mathcal{S}} \mu'$;

relative completeness: $\mu \simeq^{\mathrm{c}}_{\mathcal{S}} \mu'$ implies $\forall$ net ρ of $\mathcal{S}$, $[\![\rho]\!] \mid [\![\mu]\!] \; \tau \dot{\approx}_{\mathcal{T}} \; [\![\rho]\!] \mid [\![\mu']\!]$. This is a consequence of the (easy to verify) fact that, for any three INSs $\mathcal{S}, \mathcal{T}, \mathcal{U}$, $\mu \; \mathcal{S} \dot{\approx}_{\mathcal{T}} \; \nu$ and $\nu \; \mathcal{T} \dot{\approx}_{\mathcal{U}} \; \rho$ implies $\mu \; \mathcal{S} \dot{\approx}_{\mathcal{U}} \; \rho$.

4 Multirules Alone Do Not Give Concurrency

In the following, we fix an arbitrary INS $\mathcal{S}$.

Definition 11 (Must observability). *A port x is said to be* must-observable *in the net μ if, for all μ' s.t. $\mu \to^* \mu'$, we have $\mu' \Downarrow_x$. In that case, we write $\mu \Downarrow\!\Downarrow_x$.*

Observe that, by definition, must observability is preserved by reduction.

Lemma 1. *Let x be a port and let $\mu \equiv \mu' \mid \alpha(y; \tilde{z})$ be a net of $\mathcal{S}$, with x different from y and all of the ports in $\tilde{z}$, and $y \notin \mathrm{fp}(\mu')$. Then, $\mu \Downarrow\!\Downarrow_x$ iff $\mu' \Downarrow\!\Downarrow_x$.*

Proof. The cell $\alpha(y; \tilde{z})$ may react only on y, but y is free in μ, so the cell does not participate in any reduction of μ. □

For technical reasons, we introduce the following restricted notion of barbed bisimulation:

Definition 12 (x-bisimulation). *Let x be a port. An* x-bisimulation *is a binary relation $\mathcal{B}$ on nets of $\mathcal{S}$ such that, whenever $\mathcal{B}(\mu, \nu)$, $\mu \downarrow_x$ implies $\nu \Downarrow_x$ and $\nu \downarrow_x$ implies $\mu \Downarrow_x$, plus the usual reduction properties required by barbed bisimulations (last two points of Definition 9).*

In other words, an x-bisimulation is a usual barbed bisimulation in which we content ourselves with simulating barbs on x only.

Lemma 2. *Let $\mathcal{B}$ be an x-bisimulation, and let $\mathcal{B}(\mu, \nu)$. Then, $\mu \Downarrow\!\Downarrow_x$ iff $\nu \Downarrow\!\Downarrow_x$.*

Proof. Immediate. □

Lemma 3. *Suppose that $\mathcal{S}$ is uniport and simply-wired, and let μ be a simply-wired net of $\mathcal{S}$ such that $\mu \Downarrow\!\Downarrow_x$. Then, for every simply-wired net ν such that $x \notin \mathrm{fp}(\nu)$, $(\mu \mid \nu) \Downarrow\!\Downarrow_x$.*

Proof. By definition, $\mathcal{O}_{\mathcal{S}} \neq \emptyset$, so let $(\alpha, 1, \alpha', 1, k) \in \mathcal{O}_{\mathcal{S}}$. Let $\tilde{y}$ be a repetition-free sequence not containing x, of length equal to the number of auxiliary ports of α, and consider the relation

$$\mathcal{B} = \{(\mu \mid \nu, \alpha(x; \tilde{y})) \; ; \; \mu, \nu \text{ simply-wired}, \; \mu \Downarrow\!\Downarrow_x, \; x \notin \mathrm{fp}(\nu)\}.$$

We claim that $\mathcal{B}$ is an x-bisimulation. Let $(\mu \mid \nu, \alpha(x; \tilde{y})) \in \mathcal{B}$. First of all, $\mu \Downarrow\!\Downarrow_x$ implies $\mu \Downarrow_x$ which implies $(\mu \mid \nu) \Downarrow_x$, and by hypothesis $\alpha(x; \tilde{y}) \downarrow_x$, so the first two properties are met. Since $\alpha(x; \tilde{y})$ does not reduce, it is enough to show how $\alpha(x; \tilde{y})$ simulates a reduction $\mu \mid \nu \to \rho$. Such a reduction necessarily comes from an active pair ϕ. If ϕ is entirely contained in μ or ν, the definition of $\mathcal{B}$

allows us to conclude immediately. So we suppose that ϕ is an active pair created by the juxtaposition of μ and ν, *i.e.*, we may assume that

$$\mu \equiv \mu' \mid \beta(z;\tilde{t}),$$
$$\nu \equiv \gamma(z;\tilde{s}) \mid \nu',$$

with z free both in μ and ν, because both nets are simply-wired. Then, if ρ' is either $\beta_1 \overset{k}{\bowtie} \gamma_1\{\tilde{t}/\tilde{p},\tilde{u}/\tilde{q}\}$ or $\gamma_1 \overset{k}{\bowtie} \beta_1\{\tilde{u}/\tilde{p},\tilde{t}/\tilde{q}\}$ (for some irrelevant k), we have $\rho = \mu' \mid \rho' \mid \nu'$. But by Lemma 1, $\mu'\Downarrow_x$, so $(\rho, \alpha(x;\tilde{y})) \in \mathcal{B}$ by definition of $\mathcal{B}$.

Now, obviously $\alpha(x;\tilde{y})\Downarrow_x$ (as already observed, we have $\alpha(x;\tilde{y})\downarrow_x$ and the net does not reduce), so we may conclude by Lemma 2. □

Lemma 3 is false in presence of multiwires or multiports. For instance, consider an INS in which there are two symbols α, β, of degree 1 and 2, respectively, with the following interaction rule (which is observable, since it is the only one):

$$\alpha(x) \mid \beta(x;y) \rightarrow \alpha(y).$$

If we set $\mu = \alpha(x) \mid [x,y,z]$, we obviously have $\mu\Downarrow_y$ and $\mu\Downarrow_z$. However, for example, although still observable, z is no longer must-observable in $\mu \mid \beta(y;s)$, because $\mu \mid \beta(y;s) \rightarrow \alpha(s) \mid [z]$, in which there is no way to observe z. Similar examples may be built with multiports.

Theorem 1. *There exists an INS $\mathcal{S}$ which cannot be translated into any simply-wired, uniport INS $\mathcal{T}$ using only simply-wired nets.*

Proof. Take as $\mathcal{S}$ the system defined above, in which we allow nets containing multiwires, and suppose there exists a translation $[\![\cdot]\!]$ into a simply-wired, uniport INS $\mathcal{T}$ whose image consists of simply-wired nets only. Let $\mu = \alpha(x) \mid [x,y,z]$. Since $\mu \dot{\approx} [\![\mu]\!]$, we must have $[\![\mu]\!]\Downarrow_z$. Consider now the net $\rho = \mu \mid \beta(y;s)$. By the homomorphism property, $[\![\rho]\!] = [\![\mu]\!] \mid [\![\beta(y;s)]\!]$. By port preservation, $x \notin \mathrm{fp}([\![\beta(y;s)]\!])$, so we may apply Lemma 3 (all nets in the image of the translation are simply wired) and infer that $[\![\rho]\!]\Downarrow_z$. But we saw above that we do not have $\rho\Downarrow_z$, contradicting the fact that $\rho \dot{\approx} [\![\rho]\!]$. □

As already mentioned, although the system $\mathcal{S}$ used in the proof is uniport and uses multiwires, there is no difficulty in finding a simply-wired but multiport system $\mathcal{S}'$ for which Theorem 1 holds (with basically the same proof).

5 Comparing Multiwire and Multiport Concurrency

Definition 13 (Symmetric net). *A net μ is* strictly symmetric *if there exists a net ν whose free ports contain (but do not necessarily coincide with) $\tilde{s},\tilde{t},\tilde{u}$ such that*

$$\mu = \nu\{\tilde{a}/\tilde{s},\tilde{a}'/\tilde{t},\tilde{x}/\tilde{u}\} \mid \nu\{\tilde{a}'/\tilde{s},\tilde{a}/\tilde{t},\tilde{x}'/\tilde{u}\}.$$

In that case, the free ports of μ are $\tilde{x},\tilde{x}'$, and the pairs of ports x_i, x_i' are said to be exchanged *by the symmetry. We say that μ is* symmetric *if $\mu \equiv \mu_0$ with μ_0 strictly symmetric.*

Symmetric nets enjoy the following three fundamental properties: they are preserved by translations (if they are strict), their barbs always "come in pairs" and, if we are in a uniport system, there is no way of irreversibly breaking the symmetry in just one reduction step.

Lemma 4. *If μ is a strictly symmetric net and $[\![\cdot]\!]$ is a translation, $[\![\mu]\!]$ is strictly symmetric too.*

Proof. An immediate consequence of the homomorphism and port invariance properties of translations. □

Lemma 5. *Let μ be a symmetric net and let $x, x' \in \mathrm{fp}(\mu)$ be exchanged by the symmetry. Then:*
- *$\mu{\downarrow_x}$ iff $\mu{\downarrow_{x'}}$;*
- *$\mu{\Downarrow_x}$ iff $\mu{\Downarrow_{x'}}$.*

Proof. If $\tilde{y}, \tilde{y}'$ are the free ports of μ, with y_i, y_i' exchanged by the symmetry, then $\mu = \mu\{\tilde{y}'/\tilde{y}, \tilde{y}/\tilde{y}'\}$. The result then follows immediately. □

Lemma 6. *Let μ be a symmetric net in a uniport INS and let $\mu \to \nu$. Then, there exists a symmetric net ν' s.t. $\nu \to \nu'$.*

Proof. Let $\mu \equiv \rho \mid \rho'$, with ρ, ρ' instances of the same net ρ_0 as in Definition 13. If the active pair ϕ reduced to obtain ν is entirely in one of the two symmetric components of μ, it has a counterpart ϕ' in the other component, which obviously has a residue in ν (*cf.* Definition 8), by reducing which, in the same way as ϕ, we obtain a symmetric net ν'. Otherwise, the active pair ϕ is created by the juxtaposition of the two copies of ρ_0, so we have $\rho = \rho_1 \mid \alpha(a; \tilde{b})$, $\rho' = \beta(a; \tilde{b}') \mid \rho_1'$ and ϕ is composed by the α and β cells. But $\alpha \neq \beta$ (condition 1 of Definition 5) so we must actually have $\rho_0 = \pi \mid \alpha(s; \tilde{u}) \mid \beta(t; \tilde{v})$, *i.e.*, there is a β cell in ρ_1 and an α cell in ρ_1'. By symmetry, these form an active pair ϕ' in μ which is of the same nature as ϕ. Again, ϕ' has a residue in ν by reducing which (in the same way as ϕ) we obtain a symmetric net ν'. □

Lemma 6 is false in multiport systems. Consider the INS $\mathcal{S}$ defined as follows: its alphabet consists of two symbols α, with two principal ports and one auxiliary port, and υ, with one principal and one auxiliary port; the reduction rules are

$$\alpha(a, s; x) \mid \alpha(t, a; y) \to \upsilon(x; s) \mid [t, y],$$

and any rule for $\alpha(a, t; x) \mid \upsilon(a; y)$, which is observable. The idea is that when two α cells "meet", one on its first and the other on its second principal port, the one which interacts on the first principal port "wins". Victory is represented by the fact that its auxiliary port becomes the principal port of a υ cell, which is observable by virtue of the (otherwise irrelevant) observable rule for $(\alpha, 1, \upsilon, 1)$.

Let now

$$\mho = \alpha(a, b; x) \mid \alpha(b, a; y),$$

which is obviously strictly symmetric. We have $\mho \to \upsilon(x; y) = \mu_x$ and $\mho \to \upsilon(y; x) = \mu_y$, both of μ_x and μ_y do not reduce further and neither of them is symmetric. In fact, they are such that $\mu_x{\downarrow_x}$ but $\mu_x{\not\Downarrow_y}$, whereas $\mu_y{\downarrow_y}$ but $\mu_y{\not\Downarrow_x}$.

Theorem 2. *There exists an INS $\mathcal{S}$ which cannot be translated into any uniport INS without introducing divergence.*

Proof. We take as $\mathcal{S}$ the multiport system just introduced above and we consider the net we denoted by $\mho$. Let $[\![\cdot]\!]$ be a translation of $\mathcal{S}$ into a uniport INS and let $\nu_0 = [\![\mho]\!]$. By Lemma 4, ν_0 is symmetric and x, y are exchanged by its symmetry. By the bisimulation property, we know that there exists a barbed bisimulation $\mathcal{B}$ such that $\mathcal{B}(\mho, \nu_0)$. Since $\mho \to \mu_x$, we must have $\nu_0 \to^* \nu_x$ such that $\mathcal{B}(\mu_x, \nu_x)$. Since $\mu_x{\downarrow_x}$ but $\mu_x \not\downarrow_y$, by Lemma 5 we must have $\nu_x \neq \nu_0$, which means that at least one reduction step is possible from ν_0. Then, we may apply Lemma 6 and infer that $\nu_0 \to^* \nu_1$ in at least one reduction step, with ν_1 symmetric. But this implies that $\mho \to^* \mu_1$ such that $\mathcal{B}(\mu_1, \nu_1)$. Now, μ_1 can only be one of μ_x, μ_y or $\mho$ itself, but the symmetry of ν_1 and Lemma 5 rule out the first two cases, hence $\mathcal{B}(\mho, \nu_1)$.

The reader is invited to check that, in the above reasoning, we deduced $\mathcal{B}(\mho, \nu_1)$ starting from $\mathcal{B}(\mho, \nu_0)$ using only the fact that ν_0 is symmetric and that its two free ports are x, y (and must therefore be exchanged by its symmetry). These properties still hold for ν_1, so we may apply the reasoning again and again, obtaining a reduction sequence $[\![\mho]\!] = \nu_0 \to^* \nu_1 \to^* \nu_2 \to^* \cdots$, in which every reduction $\nu_i \to^* \nu_{i+1}$ is of length at least 1, so $[\![\mho]\!]$ diverges. □

6 Discussion

Significance. A potentially controversial point of our definition of barb is its parametricity, which makes barbed congruence somewhat arbitrary. A possible answer is the following: there is always a default choice, which consists in deeming *every* rule observable. Concretely, a default barb $\mu{\downarrow_x}$ is equivalent to the fact that x is a free principal port in μ, *i.e.*, $\mu = \alpha(\tilde{x}; \tilde{y}) \mid \mu'$ with $x = x_i$ for some i and α.

Default barbed congruence is analogous to usual barbed congruence in standard process calculi, including the solos calculus [14], of which interaction nets are strongly reminiscent.[3] The possibility of using smaller sets of observable rules should only be considered in encodings: if an INS $\mathcal{S}$, with its default barbed congruence, is to be encoded in an INS $\mathcal{T}$, it may be reasonable to consider instead $(\Sigma_\mathcal{T}, \bowtie_\mathcal{T}, \mathcal{O})$, where we exclude from $\mathcal{O}$ the "administrative" rules of $\mathcal{T}$, thus weakening barbed congruence (in the extreme case $\mathcal{O} = \emptyset$, which is not allowed by our definition, barbed congruence would equate everything).

It is also interesting to consider default barbed congruence in Lafont interaction nets systems, *i.e.*, the strictly deterministic kind, for which definitions of observational equivalences already exist [6]. In this setting, we are able to prove that, if we ignore the notion of "constructor symbol" used by Fernández and Mackie (which has no counterpart in our definitions), default barbed congruence coincides with their observational equivalence. So, at least in the simple

[3] In fact, by considering two families of symbols ι_n, o_n with the rules $\iota_n(x; \tilde{y}) \mid o_n(x; \tilde{z}) \to [y_1, z_1] \mid \cdots \mid [y_n, z_n]$, one basically obtains the replication-free solos calculus with explicit fusions [21], with names represented by multiwires.

deterministic case, our definitions fall back on something already known to be meaningful.

As far as our notion of translation is concerned, it is based on properties which are mostly agreed upon in the literature [8,20]. The only other existing notion of translation for interaction nets is the one mentioned above, formulated by Lafont for his deterministic systems [10]. It is possible to show that, if we consider default barbed congruence, a Lafont's translation induces a translation in our sense. Conversely, thanks to determinism, our definition does not differ from Lafont's one in an essential way, although it is more permissive.

Turning to our separation results, as they are technically formulated, Theorem 1 and Theorem 2 may be easily criticized: even if we agree on the reasonability of our notion of translation, the sole existence of an untranslatable system may not be enough to convincingly separate two extensions; it all depends on the relevance of such a system.

We believe that the relevance of the untranslatable systems is given by Lemma 3, for Theorem 1, and Lemma 6, for Theorem 2. In both cases, there is one "limiting" property which always holds in one extension of interaction nets but fails in the "more expressive" ones. In the first case, the limitation is so severe that we are led to conclude that multirules alone cannot express concurrency: indeed, Lemma 3 is false in any standard process calculus. On the other hand, Lemma 6 shows the same limitation pointed out by Palamidessi for the asynchronous π-calculus: in absence of multiports, interaction nets are unable to make certain irreversible choices in just one step (*i.e.*, synchronously). Instead, such choices must always involve a reversible "pre-commitment" phase. Only once such a phase is successfully concluded may the choice be irreversibly committed. Theorem 2 shows this for an "electoral system" with only two nodes, but the argument scales up to arbitrarily large "leader-election" nets, as in [19].

Concluding Remarks. We observe that our first separation result casts doubts on the value of the encoding of the π-calculus in differential interaction nets (which are a multirule INS) given in [4]. As already pinpointed by the second author [18], that encoding supposes a labelled semantics of differential interaction nets which is not "realistic" in terms of concurrency.

We are also left with an interesting open question concerning the relaxation of condition 1 of Definition 5, allowing "self-interaction", as considered by Lafont [12]. We know that Lemma 6 fails in presence of such a relaxation. In process calculi, this would correspond to introducing "neutral" prefixes, neither input nor output, which may synchronously interact with each other. Palamidessi's symmetry argument then does not apply straightforwardly and "neutral" synchronization might be as expressive as input/output synchronization.

Acknowledgments. This work was partially supported by ANR projects Panda (09-BLAN-0169-02) and Logoi (10-BLAN-0213-02). The second author wishes to dedicate this paper to the memory of Jamey Avon.

References

1. Alexiev, V.: Non-deterministic Interaction Nets. Ph.D. Thesis, University of Alberta (1999)
2. Asperti, A., Guerrini, S.: The Optimal Implementation of Functional Programming Languages. Cambridge University Press (1998)
3. Beffara, E., Maurel, F.: Concurrent nets: A study of prefixing in process calculi. Theoretical Computer Science 356(3), 356–373 (2006)
4. Ehrhard, T., Laurent, O.: Interpreting a Finitary Pi-Calculus in Differential Interaction Nets. Information and Computation 208(6), 606–633 (2010)
5. Ehrhard, T., Regnier, L.: Differential Interaction Nets. Theoretical Computer Science 364(2), 166–195 (2006)
6. Fernández, M., Mackie, I.: Operational Equivalence for Interaction Nets. Theoretical Computer Science 297(1-3), 157–181 (2003)
7. Gonthier, G., Abadi, M., Lévy, J.J.: The geometry of optimal lambda reduction. In: Sethi, R. (ed.) Proceedings of POPL, pp. 15–26. ACM Press (1992)
8. Gorla, D.: Towards a unified approach to encodability and separation results for process calculi. Information and Computation 208(9), 1031–1053 (2010)
9. Honda, K., Laurent, O.: An exact correspondence between a typed pi-calculus and polarised proof-nets. Theor. Comput. Sci. 411(22-24), 2223–2238 (2010)
10. Lafont, Y.: Interaction nets. In: Allen, F.E. (ed.) Proceedings of POPL, pp. 95–108. ACM Press (1990)
11. Lafont, Y.: From proof nets to interaction nets. In: Girard, J.Y., Lafont, Y., Regnier, L. (eds.) Advances in Linear Logic, pp. 225–247. Cambridge University Press (1995)
12. Lafont, Y.: Interaction Combinators. Information and Computation 137(1), 69–101 (1997)
13. Lamping, J.: An algorithm for optimal lambda calculus reduction. In: Allen, F.E. (ed.) Proceedings of POPL, pp. 16–30. ACM Press (1990)
14. Laneve, C., Victor, B.: Solos In Concert. Mathematical Structures in Computer Science 13(5), 657–683 (2003)
15. Mackie, I.: An Interaction Net Implementation of Additive and Multiplicative Structures. Journal of Logic and Computation 15(2), 219–237 (2005)
16. Mackie, I.: An interaction net implementation of closed reduction. In: Scholz, S.-B., Chitil, O. (eds.) IFL 2008. LNCS, vol. 5836, pp. 43–59. Springer, Heidelberg (2011)
17. Mazza, D.: Multiport Interaction Nets and Concurrency. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 21–35. Springer, Heidelberg (2005)
18. Mazza, D.: The True Concurrency of Differential Interaction Nets. Mathematical Structures in Computer Science (to appear, 2013) (accepted for publication)
19. Palamidessi, C.: Comparing the expressive power of the synchronous and asynchronous pi-calculi. Math. Structures Comput. Sci. 13(5), 685–719 (2003)
20. Parrow, J.: Expressiveness of Process Algebras. Electronic Notes in Theoretical Computer Science 209, 173–186 (2008)
21. Wischik, L., Gardner, P.: Explicit fusions. Theoretical Computer Science 340(3), 606–630 (2005)
22. Yoshida, N.: Graph notation for concurrent combinators. In: Ito, T. (ed.) TPPP 1994. LNCS, vol. 907, pp. 393–412. Springer, Heidelberg (1995)

An Epistemic Perspective on Consistency of Concurrent Computations

Klaus von Gleissenthall[1] and Andrey Rybalchenko[1,2]

[1] Technische Universität München
[2] Microsoft Research Cambridge

Abstract. Consistency properties of concurrent computations, e.g., sequential consistency, linearizability, or eventual consistency, are essential for devising correct concurrent algorithms. In this paper, we present a logical formalization of such consistency properties that is based on a standard logic of knowledge. Our formalization provides a declarative perspective on what is imposed by consistency requirements and provides some interesting unifying insight on differently looking properties.

1 Introduction

Writing correct distributed algorithms is notoriously difficult. While in the sequential case, various techniques for proving algorithms correct exist [14, 17], in the concurrent setting, due to the nondeterminism induced by scheduling decisions and transmission failures, it is not even obvious what correctness actually means. Over the years, a variety of different *consistency properties* restricting the amount of tolerated nondeterminism have been proposed [10–12, 16, 18, 19]. These properties range from simple properties like sequential consistency [16] or linearizability [10, 12] to complex conditions like eventual consistency [19], a distributed systems condition. Reasoning about these properties is a difficult, yet important task since their implications are often surprising.

Currently, the study of consistency properties and the development of reasoning tools and techniques for such properties [4, 6, 20] is done for each property individually, i.e., on a per property basis. To some extent, this trend might be traced back to the way consistency properties are formulated. Typically, they explicitly require existence of certain computation traces that are obtained by rearrangement of the trace that is to be checked for consistency, i.e., these descriptions of consistency properties do not rely on a logical formalism. While such an approach provides fruitful grounds for the design of specialized algorithms and efficient tools, it leaves open important questions such as how various properties relate to each other or whether advances in dealing with one property can be leveraged for dealing with other properties.

In contrast to the trace based definitions found in literature, we propose to study consistency conditions in terms of epistemic logic [7, 9]. Here we can rely on a distributed knowledge modality [7], which is a natural fit for describing distributed computation. In this logic, an application $D_G(\varphi)$ of the distributed

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 212–226, 2013.

knowledge modality to a formula φ denotes the fact that a group G *knows* that a formula φ holds.

We present a logical formalization of three consistency properties: the classical sequential consistency [16] and linearizability [10, 12], as well as a recently proposed formulation [5] of eventual consistency for distributed databases [19]. Our characterizations show that moving the viewpoint from reasoning about traces (models) to reasoning about knowledge (logic) can lead to new insights. When formulated in the logic of knowledge, these differently looking properties agree on a common schematic form: $\neg D_G(\neg \mathit{correct})$. According to this schematic form, a computation satisfies a consistency property if and only if a group G of its participants, i.e., threads or distributed nodes, *do not know that* the computation violates a specification *correct* that describes computations from the *sequential* perspective, i.e., without referring to permutations thereof. For example, when formalising sequential consistency of a concurrent register *correct* only states that the first read operation returns zero and any subsequent read operation returns the value written by the latest write operation.

The common form of our characterizations exposes the differences between the consistency properties in a formal way. A key difference lies in the group of participants that provides knowledge for validating the specification *correct*. For example, a computation is sequentially consistent if it satisfies the formula $\neg D_{\textsc{Threads}}(\neg \mathit{correct})$, i.e., the group G of agents needed to validate the sequential specification comprises the group of threads Threads accessing the shared memory. Surprisingly, the same group of agents is needed to validate eventual consistency, since in our logic it is characterized by the formula $\neg D_{\textsc{Threads}}(\neg \mathit{correctEVC})$. This reveals an insight that eventual consistency is actually not an entirely new consistency condition, but rather an instance of sequential consistency that is determined by a particular choice of *correct*. In contrast to the two above properties, the threads' knowledge is not enough to validate linearizability. To capture linearizability, the set of participants G needs to go beyond the participating threads Threads and include an additional observer thread *obs* as well. The observer only acquires knowledge of the relative order between returns and calls. As logical characterization of linearizability, we obtain $\neg D_{\textsc{Threads}\cup\{obs\}}(\neg \mathit{correct})$.

We show that including the observer induces a different kind of knowledge, i.e., it weakens the modal system from *S5* to *S4* [21]. As a consequence, the agents lose certainty about their decision whether or not a trace is consistent. For sequential consistency (*seqCons*) the agents know whether or not a trace is sequentially consistent, i.e., the formula $(\mathit{seqCons} \leftrightarrow D_{\textsc{Threads}}(\mathit{seqCons})) \wedge (\neg \mathit{seqCons} \leftrightarrow D_{\mathit{Threads}}(\neg \mathit{seqCons}))$ is valid. In contrast, for linearizability (*Lin*) the threads cannot be sure whether a trace they validate as linearizable is indeed linearizable, i.e., there exists a trace that satisfies $\mathit{Lin} \wedge \neg D_{\textsc{Threads}\uplus\{obs\}}(\mathit{Lin})$.

The discovery that eventual consistency can be reduced to sequential consistency is facilitated by a generalization of classical sequential consistency that follows naturally from taking the epistemic perspective. Our formalization of *correct* for eventual consistency is given by *correctEVC* that requires nodes to

keep consistent logs, i.e., whenever a transaction is received by a distributed node, the transaction must be inserted into the node's logs in a way that is consistent with the other nodes' recordings. We allow *correctEVC* not only to refer to events that are performed by the nodes that take part in the computation, but also to auxiliary events that model the *environment* that interacts with nodes. We use the environment to model transmission of updates from one distributed node to another. Our knowledge characterization then implicitly quantifies over the order of occurrence of such events, which serves as a correctness certificate for a given trace.

Contributions. In summary, our paper makes the following contributions. We provide characterizations for sequential consistency (Section 4), eventual consistency (Section 5) and linearizability (Section 6) which we prove correct wrt. their standard definitions. Our characterizations reveal a remarkable similarity between consistency properties that is not apparent in their standard formulations. Through our characterizations, we identify a natural generalization of sequential consistency that allows us to reduce eventual consistency, a complex property usually defined by the existence of two partial-ordering relations, to sequential consistency. In contrast to this reduction, we show that linearizability requires a different kind of knowledge than sequential consistency and prove a theorem (Section 7) illustrating the ramifications of this difference.

2 Examples

In this section, before providing technical details, we give an informal overview of our characterizations.

2.1 Sequential Consistency

Trace-Based Definition. The most fundamental consistency condition that concurrent computations are intuitively expected to satisfy is sequential consistency [16]. Its original definition reads:

> The result of any execution is the same as if the operations of all the processors were executed in *some sequential order*, and the operations of each individual processor appear in the sequence in the *order specified by its program.*

Equivalently, this more formal version can be found in the literature (cf., [1]): For a trace E to be sequentially consistent, it needs to satisfy two conditions: (1) E must be *equivalent* to a witness trace E' and (2) trace E' needs to be *correct* with respect to some specification. To be equivalent, two traces need to be permutations that preserve the local order of events for each thread.

Example 1. Consider the following traces representing threads t_1 and t_2 storing and loading values on a shared register. For the purpose of this example, we

assume the register to be initialized with value 0. We use ":=" to abbreviate "equals by definitions".

$$E_1 := (t_2, ld(0))\ (t_2, ld(1))\ (t_1, st(1))$$
$$E_2 := (t_2, ld(0))\ (t_1, st(1))\ (t_2, ld(1))$$
$$E_3 := (t_2, ld(0))\ (t_1, st(1))\ (t_2, ld(2))\ .$$

Trace E_1 is sequentially consistent, because it is equivalent to E_2 and E_2 meets the specification of a shared register, i.e., each load returns the last value stored. In contrast, E_3 is not sequentially consistent, because no appropriate witness can be found. In no equivalent trace, t_2's load of 2 is preceded by an appropriate store operation.

Logic. In this paper, in contrast to the above trace-based formulation, we investigate consistency from the perspective of *epistemic logic*. Epistemic logic is a formalism used for reasoning about the knowledge distributed nodes/threads acquire in a distributed computation. For example, in trace E_1 thread t_2 *knows* it first loaded value 0 and then value 1 while t_1 *knows* it stored 1. When we consider the knowledge acquired by the threads t_1 and t_2 together as a group, we say that the group of threads $\{t_1, t_2\}$ *jointly knows* t_2 first loaded 0 and then 1 while t_1 stored 1. We denote the fact that a group G jointly knows that a formula φ holds by $D_G(\varphi)$, which is an application of the distributed knowledge modality. According to our logical characterization of sequential consistency: $\neg D_{\text{THREADS}}(\neg \mathit{correct})$, a trace is sequentially consistent, if the group of all threads accessing the shared data-structure does not jointly know that the trace is not correct.

Example 1 (continued). This means trace E_1 is sequentially consistent. In trace E_1, the threads know that t_2 first loaded 0 and then 1 and that t_1 stored 1, however they do not know in which order these events were scheduled. This means, for all they know t_1 could have stored 1 before t_2 loaded it and after t_2 loaded 1, which would meet the specification. In contrast, E_3 is not sequentially consistent. The threads know that t_2 loaded 2, however they also know that no thread stored this value. This means E_3 cannot have met the specification.

Indistinguishability. We formalize this notion of knowledge in terms of the local perspective individual threads have on the computation. We extract this perspective by a function $\downarrow$ such that $E \downarrow t$ projects trace E onto the local events of thread t. If two traces do not differ from the local perspective of thread t, we say that they are indistinguishable for t. We write $E \sim_t E'$ to denote that for thread t, trace E is indistinguishable from trace E'. Combining their abilities to distinguish traces, a group of threads can distinguish two traces whenever there is a thread in the group that can. We write $E \sim_G E'$ to denote that for for every member of group G trace E is indistinguishable from trace E'. Indistinguishability allows us to define the knowledge of a group. A group G knows a fact φ if this fact holds on all traces that the threads in G cannot distinguish from the

actual trace. We write $E \vDash \varphi$ to say that trace E satisfies formula φ. Formally (see Section 3.3): $E \vDash D_G(\varphi)$:iff for all E' s.t. $E \sim_G E'$: $E' \vDash \varphi$, where we use ":iff" to abbreviate "equals by definition".

Example 1 (continued). For trace E_1, the thread-local projections are: $E_1 \downarrow t_1 = (t_1, st(1))$ and $E_1 \downarrow t_2 = (t_2, ld(0))(t_2, ld(1))$. We get the same projections for E_2, and $E_3 \downarrow t_1 = (t_1, st(1))$ and $E_3 \downarrow t_2 = (t_2, ld(0))(t_2, ld(2))$. From these projections, we get: $E_1 \sim_{t_1} E_2 \sim_{t_1} E_3$ and $E_1 \sim_{t_2} E_2$ but $E_1 \not\sim_{t_2} E_3$ and $E_2 \not\sim_{t_2} E_3$. For groups of threads, we have: $E_1 \sim_{\{t_1,t_2\}} E_2$ but $E_2 \not\sim_{\{t_1,t_2\}} E_3$, because $E_2 \not\sim_{t_2} E_3$. We write $E \vDash \mathit{correctREG}$ to say E is correct with respect to the specification of a shared register. Then $E_1 \vDash \neg D_{\text{Threads}}(\neg \mathit{correctREG})$, $E_2 \vDash \neg D_{\text{Threads}}(\neg \mathit{correctREG})$ and $E_3 \vDash D_{\text{Threads}}(\neg \mathit{correctREG})$.

Knowledge in the Trace-Based Formulation. Interestingly, the notion of equivalence found in the trace-based formulation of sequential consistency precisely corresponds to $\sim_{\text{Threads}}$, the indistinguishability relation of all threads accessing the shared data-structure. This suggests that the knowledge-based formulation of consistency lies already buried in the original definition. Similarly, the formulation "The result of any execution is the same as if ...", found in the original definition alludes to the possibility of a fact φ, which, in epistemic logic, is represented by the dual modality of knowledge $\neg D_G(\neg\varphi)$.

2.2 Eventual Consistency

Eventual consistency [19] is a correctness condition for distributed database systems, as those employed in modern geo-replicated internet services. In such systems, threads (distributed nodes) keep local working-copies (repositories) of the database which they may update by performing a commit operation. Queries and updates have revision ids, representing the current state of the local copy. Whenever a thread commits, it broadcasts local changes to its repository and receives changes made by other threads. After the commit, a new revision id is assigned. As the underlying network is unreliable, committed changes may however be delayed or lost before reaching other threads.

In this setting, weaker guarantees on consistency than in a multi-processor environment are required, as network partitions are unavoidable, causing updates to be delayed or lost. Consequently, eventual consistency is a prototypical example for what is called "weak"-consistency. We present a recent, partial-order based definition drawn from the literature [5] in Section 5.

Taking the knowledge perspective on eventual consistency reveals a remarkable insight. Eventual consistency is actually not an entirely new, weaker consistency condition, but sequential consistency – with an appropriate sequential specification.

In our logical characterization, eventual consistency is defined by the formula: $E \vDash \neg D_{\text{Threads}}(\neg \mathit{correctEVC})$. That is, to be eventually consistent, a trace needs to be sequentially consistent with respect to a sequential specification $\mathit{correctEVC}$. Our formula for $\mathit{correctEVC}$ uses the past time modality

$\boxminus(\varphi)$ (see Section 3.3), representing the fact that so far, formula φ was true. We specify *correctEVC* by:

$$\begin{aligned} \mathit{correctEVC} \coloneqq\ & \forall t \forall q \forall r(\boxminus(\mathit{query}(t,q,r) \rightarrow \exists \mathcal{L}(\mathcal{L}\ \mathit{validLog}\ t\ \wedge \mathit{result}(q,\mathcal{L},r)))) \\ & \wedge \mathit{atomicTrans} \wedge \mathit{alive} \wedge \mathit{fwd}\ . \end{aligned}$$

This formula says that for all threads, queries and results, so far, whenever a thread t posed a query q to its local repository, producing result r, thread t must be able to present a valid log $\mathcal{L}$, such that the result of posing query q on a machine that performed only the operations logged in log $\mathcal{L}$ matches the recorded result r. The additional conjuncts *atomicTrans*, *alive* and *fwd* specify further requirements on the way updates may be propagated in the network.

In our characterization, a log $\mathcal{L}$ is a sequence of actions (i.e., queries, updates and commits). The formula *validLog* describes the conditions a log has to satisfy to be valid for a thread t:

$$\mathcal{L}\ \mathit{validLog}\ t\ \coloneqq\ \forall a(a\ \mathit{in}\ \mathcal{L} \leftrightarrow t\ k_{log}\ a)\ \wedge \mathit{consistent}(\mathcal{L})\ .$$

This formula requires that for all actions a, a is logged in $\mathcal{L}$ (represented by the infix-predicate *in*) if and only if thread t knows about action a. A thread knows about all the actions that it performed itself and the actions performed in revisions that were forwarded to it. The formula $\mathit{consistent}(\mathcal{L})$ ensures that all actions in the log $\mathcal{L}$ appear in an order consistent with the actual order of events.

Environment Events. To make this result possible, we make a generalization that comes naturally in the knowledge setting. We allow traces to contain *environment* events that represent actions that are not controlled by the threads that participate in the computation. In our characterization, environment events are used to mark positions where updates were successfully forwarded from one client to another. By allowing *correctEVC* to refer to those events, we implicitly encode an existential quantification over all possible positions for these events. That means a trace is eventually consistent if any number of such events could have occurred such that the specification *correctEVC* is met.

Example 2. Consider the following traces of a simple database that allows clients to update and query the integer variable x:

$$\begin{aligned} E_4 \coloneqq\ & (t_1, \mathit{up}(0, x \coloneqq 0))\ (t_1, \mathit{com}(0))\ (t_1, \mathit{up}(1, x \coloneqq 1))(t_1, \mathit{com}(1)) \\ & (t_2, \mathit{qu}(0, x, 0))(t_2, \mathit{com}(0))\ (t_2, \mathit{qu}(1, x, 1)) \\ E_5 \coloneqq\ & (t_1, \mathit{up}(0, x \coloneqq 0))\ (t_1, \mathit{com}(0))\ (\mathit{env}, \mathit{fwd}(t_1, t_2, 0))(t_1, \mathit{up}(1, x \coloneqq 1)) \\ & (t_1, \mathit{com}(1))\ (t_2, \mathit{qu}(0, x, 0))(t_2, \mathit{com}(0))\ (\mathit{env}, \mathit{fwd}(t_1, t_2, 1)) \\ & (t_2, \mathit{qu}(1, x, 1))\ . \end{aligned}$$

Updates are of the form $\mathit{up}(\mathit{id}, u)$, where id is the revision-id and u the actual update. In our example, updates are variable assignments $x \coloneqq v$ meaning that a variable x is assigned value v. Queries are of the form $\mathit{qu}(\mathit{id}, q, r)$, where id

stands for the revision-id, q for the query, and r for the result. Queries in our example consist only of variables, i.e., a query returns the current value assigned. The action $com(id)$ represents the act of committing, that is, sending revision id over the network and checking for updates. Forwarding actions are performed by the environment env. The event $(env, fwd(t, t', id))$ represents the environment forwarding the changes made in revision id from thread t to thread t'.

In trace E_5, when thread t_2 queries the value of x in revision 0, thread t_2 can present the log $\mathcal{L} \coloneqq up(0, x \coloneqq 0)\ com(0)\ qu(0, x, 0)$ as an evidence of the correctness of the result 0. As by the time of t's query, only revision 0 has been forwarded from t_1 to t_2, thread t_2 only knows about t_1's first update and its own query. Querying x after the update $x \coloneqq 0$ yields 0, so $result(x, \mathcal{L}, 0)$ holds.

When thread t_2 queries x in revision 1, thread t_1's second update has been forwarded, so t_2 can present the log $\mathcal{L} \coloneqq up(0, x \coloneqq 0)\ com(0)\ up(1, x \coloneqq 1)\ com(1)\ qu(0, x, 0)\ com(0)\ qu(1, x, 1)$. Since t_2 received the t_1's revision 1 the log contains the second update $x \coloneqq 1$ and t_2's query of x returns 1. This means $E_5 \vDash correctEVC$. As a consequence, we have $E_4 \vDash \neg D_{\textsc{Threads}}(\neg correctEVC)$, because $E_4 \sim_{\textsc{Threads}} E_5$ and $E_5 \vDash correctEVC$. The forwarding events in E_5 mark positions where the transmission of updates through the network could have occurred to make the computation meet $correctEVC$.

2.3 Linearizability

While the threads' knowledge characterizes sequential consistency and eventual consistency, their knowledge is not strong enough to define linearizability. Linearizability extends sequential consistency by the requirement that method calls must effect all visible change of the shared data at some point between their invocation and their return. Such a point is called the *linearization points* of the method.

To characterize linearizability, we introduce another agent called *the observer* that tracks the available information on linearization points. To do this, the observer monitors the order of non-overlapping (sequential) method calls in a trace. The observer's view of a trace is the order of non-overlapping method calls. This order is represented by a set of pairs of return and invoke events, such that the return took place before the invocation. We extract this order by a projection function $obs(\cdot)$.

Example 3. Consider the following traces where method calls are split into invocation- and return events:

$$E_6 \coloneqq (t_2, inv\ ld())\ (t_2, ret\ ld(1))\ (t_1, inv\ st(1))\ (t_1, ret\ st(true))$$
$$E_7 \coloneqq (t_2, inv\ ld())\ (t_1, inv\ st(1))\ (t_2, ret\ ld(1))\ (t_1, ret\ st(true))$$
$$E_8 \coloneqq (t_1, inv\ st(1))\ (t_1, ret\ st(true))\ (t_2, inv\ ld())\ (t_2, ret\ ld(1))\ .$$

For trace E_6, the observer's projection function $obs(\cdot)$ yields: $obs(E_6) = \{(\ (t_2, ret\ ld(1)), (t_1, inv\ st(1))\ )\}$. This means the observer sees that t_2's load returned before t_1's store was invoked. In trace E_7, the method calls overlap.

Consequently, the observer knows nothing about this trace: $obs(E_7) := \varnothing$. For E_8, we get $obs(E_8) = \{(\,(t_1, ret\ st(true)), (t_2, inv\ ld())\,)\}$.

The observer's view tracks the available information on linearization points. In trace E_6, thread t_2's linearization point for the call to load must have occurred before the linearization point of t_1's call to store. This follows from the fact that $t2$'s load returned before $t1$'s call to store and that linearization point must occur somewhere between a method's invocation and its return. In trace E_7 linearization points may have occurred in any order as the method calls overlap.

To the observer, a trace E is indistinguishable from a trace E' if the order of linearization points in E is preserved in E' and maybe an order between additional linearization points is fixed (see Section 3.1): $E \preceq_{obs} E'$:iff $obs(E) \subseteq obs(E')$. A trace E is linearizable if the threads together with the observer do not know that the trace is incorrect: $E \vDash \neg D_{\text{THREADS}\uplus\{obs\}}(\neg correct)$.

Example 3 (Continued). We have $E_6 \not\preceq_{obs} E_7$, but $E_7 \preceq_{obs} E_6$. Trace E_7 is linearizable since $E_7 \sim_{\text{THREADS}\uplus\{obs\}} E_8$ and $E_8 \vDash correctREG$. However, trace E_6 is not linearizable since there is no indistinguishable trace that meets the specification. Note that the threads without the observer could not have detected this violation of the specification, i.e., $E_6 \vDash \neg D_{\text{THREADS}} \neg correctREG$.

2.4 Knowledge about Consistency

As we describe sequential consistency in a standard logic of knowledge, corresponding axioms apply (see, e.g. [21, chapter 2.2]). For example, everything a group of threads knows is also true: (T) := $\vDash D_G(\varphi) \rightarrow \varphi$ (Truth axiom), groups of threads know what they know: (4) := $\vDash D_G(\varphi) \rightarrow D_G(D_G(\varphi))$ (positive introspection) and groups of threads know what they do not know: (5) := $\vDash \neg D_G(\varphi) \rightarrow D_G(\neg D_G(\varphi))$ (negative introspection). For a complete axiomatization of a similar epistemic logic with time see [3].

Interestingly, adding the observer not only strengthens the threads' ability to distinguish traces but changes the *kind* of knowledge agents acquire about a computation. Whereas $\sim_{\text{THREADS}}$ is an *equivalence relation*, $\sim_{\text{THREADS}\uplus\{obs\}}$ is only a *partial order*. As a consequence, D_{THREADS} corresponds to the modal system *S5*, whereas $D_{\text{THREADS}\uplus\{obs\}}$ corresponds to the weaker system *S4* [21]. This means, that $D_{\text{THREADS}\uplus\{obs\}}$ does not satisfy the axiom of negative introspection (5).

It seems natural to ask if the differences in the type of knowledge between sequential consistency and linearizability affect the ability to detect violations of the specification. In Section 7, we show that the difference the lack of axiom (5) makes, lies in the certainty threads have about their decision. Whereas for sequentially consistent ($seqCons := \neg D_{\text{THREADS}}(\neg correct)$), whenever the threads decide that a trace is sequentially consistent, they can be sure that the trace is indeed sequentially consistent: ($seqCons \leftrightarrow D_{\text{THREADS}}(seqCons)$) for linearizability ($Lin := \neg D_{\text{THREADS}\uplus\{obs\}}(\neg correct)$), it can occur that the threads together with the observer decide that a trace is linearizable, however, they cannot be sure that it really was: $Lin \wedge \neg D_{\text{THREADS}\uplus\{obs\}}(Lin)$.

3 Logic of Knowledge

In this section we present a standard logic of knowledge (see [9]) that we use for our characterizations. We follow the exposition of [15]. We define the set $\mathcal{E}$ of events as $\mathcal{E} \ni e := (t, act)$, representing $t \in \textsc{Threads} \uplus \{env\}$ performing an action $act \in \mathcal{A}$. The environment env can perform synchronization events that go unseen by the threads. In our characterization of eventual consistency, the environment forwards transactions from one node to the other. We define the generic set of actions: $\mathcal{A} \ni act := \ inv(m, v) \mid ret(m, v)$. Threads can invoke or return from methods $m \in \textsc{Methods}$ with $v \in \textsc{Values}$. For our characterization of eventual consistency, we instantiate $\mathcal{A}$ with application-specific actions. These can easily be translated back into the generic form by splitting up events into separate invocation- and return-parts.

3.1 Preliminaries

We denote by $\mathcal{E}^*$ the set of finite-, and by $\mathcal{E}^\infty$ the set of infinite sequences over $\mathcal{E}$. We denote the empty sequence by ϵ. Let $\mathcal{E}^\omega := \mathcal{E}^* \uplus \mathcal{E}^\infty$ and $E \in \mathcal{E}^\omega$. Then $E \downharpoonright i$ denotes the finite prefix up to- and including i. We let $E@i$ be the element of sequence E at position i. We define $len(E)$ to be the length of E, where $len(\epsilon) = 0$, and $len(E) = \omega$, if $E \in E^\infty$. For $e \in \mathcal{E}$, we say that $pos(e, E) = j$, if $E@j = e$ and $pos(e, E) = \omega$ otherwise. Hence, we write $e \in E$ if $pos(e, E) < \omega$. We make the assumption that each event occurs only once in a trace. This is not a restriction as we could add a unique time-stamp or a sequence number to each event.

We formally define projection functions and indistinguishability relations. A thread's view of a computation trace is the part of the trace it can observe. We define this part by a projection function that extracts the respective events. We use this projection function to define an indistinguishability relation for each thread.

Thread Indistinguishability Relation. For a thread $t \in \textsc{Threads}$ the indistinguishability relation $\sim_t \subseteq (\mathcal{E}^\omega \times (\mathbb{N} \uplus \{\omega\}))^2$ is defined such that: $(E, i) \sim_t (E', i')$:iff $(E \downharpoonright i) \downarrow t = (E' \downharpoonright i') \downarrow t$ where $\downarrow$: $(\mathcal{E}^\omega \times \textsc{Threads}) \to \mathcal{E}^\omega$ designates a projection function onto t's local perspective. $E \downarrow t$ is the projection on events in the set $\{(t, act) \mid act \in \mathcal{A}\}$, i.e., the sequence obtained from E by erasing all events that are not in the above set.

Observer Indistinguishability Relations. The observer's view of a trace is the order of non-overlapping method calls. We let $\textsc{Inv} \ni in := (t, inv(m, v))$ and $\textsc{Ret} \ni r := (t, ret(m, v))$. The indistinguishability relation of the observer $\preceq_{\text{obs}} \subseteq (\mathcal{E}^\omega \times \mathbb{N})^2$ is given by: for all $(E, i), (E', i') \in \mathcal{E}^\omega \times \mathbb{N}$: $(E, i) \preceq_{\text{obs}} (E', i')$:iff $\text{obs}(E, i) \subseteq \text{obs}(E', i')$ where obs: $(\mathcal{E}^\omega \times \mathbb{N}) \to \mathcal{P}(\mathcal{E}^2)$ designates a projection onto the observer's local view, such that: $obs(E, i) = \{(r, in) \in \textsc{Ret} \times \textsc{Inv} \mid \text{pos}(r, E) < \text{pos}(in, E) \leq i\}$. We abbreviate $obs(E) := obs(E, len(E))$.

Joint Indistinguishability Relations. Joint indistinguishability relations link pairs of traces that a group of threads can distinguish if they share their knowledge. Whenever a thread in the group can tell the difference between two traces, the group can. Let $G \subseteq \textsc{Threads}$. We define the joint indistinguishability relation of group G to be $\sim_G := (\bigcap_{t \in G} \sim_t)$ and $\sim_{G \uplus \{obs\}} := \sim_G \cap \preceq_{obs}$. For any indistinguishability relation $\sim$, we write $E \sim E'$ as an abbreviation for $(E, len(E)) \sim (E, len(E'))$.

3.2 Syntax

A formula in the logic takes the form:

$$\varphi, \psi ::= \varphi \wedge \psi \mid \neg\varphi \mid \varphi S \psi \mid \varphi U \psi \mid D_G(\varphi) \mid \forall x(\varphi) \ .$$

with $G \subseteq \textsc{Threads} \uplus \{obs\}$ and $p \in \textsc{Predicates}$, which we instantiate for each of our characterizations. The logic provides the temporal modalities $\varphi S \psi$ representing the fact that since ψ occurred, φ holds and the modality $\varphi U \psi$ representing the fact that until ψ occurrs, φ holds. Additionally, it provides the *distributed knowledge* modality D_G and first order quantification. Let Φ denote the set of all formulae in the logic.

3.3 Semantics

We now define the satisfaction relation $\vDash \subseteq (\mathcal{E}^\omega \times (\mathbb{N} \uplus \omega)) \times \Phi$. We let:

$$\begin{array}{ll} (E,i) \vDash \varphi \wedge \psi & :\text{iff } (E,i) \vDash \varphi \text{ and } (E,i) \vDash \psi \\ (E,i) \vDash \neg\varphi & :\text{iff not } (E,i) \vDash \varphi \ . \end{array}$$

We define the temporal modalities by:

$$\begin{array}{lll} (E,i) \vDash \varphi S \psi & :\text{iff} & \text{there is } j \leq i \text{ s.t. } (E,j) \vDash \psi \text{ and} \\ & & \text{for all } j < k \leq i : (E,k) \vDash \varphi \\ (E,i) \vDash \varphi U \psi & :\text{iff} & \text{there is } j \leq i \text{ s.t. } (E,j) \vDash \psi \text{ and} \\ & & \text{for all } 1 \leq k < j : (E,k) \vDash \varphi \ . \end{array}$$

We define distributed knowledge as: $(E,i) \vDash D_G(\varphi)$:iff for all (E',i'): if $(E,i) \sim_G (E',i')$ then $(E',i') \vDash \varphi$, with $G \subseteq \textsc{Threads} \uplus \{obs\}$. Let D be the domain of quantification. We define first-order quantification: $(E,i) \vDash \forall x(\varphi)$:iff for all $d \in \mathrm{D} : (E,i) \vDash \varphi[d/x]$. By $\varphi[d/x]$, we denote the term φ with all occurrences of x replaced by d. We define D as the disjoint union of all quantities used in the definition of a condition. We write $E \vDash \varphi$ as an abbreviation for $(E, len(E)) \vDash \varphi$.

Additional Definitions. For convenience, we define the following standard operators in terms of our above definitions: $\varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi)$, $\varphi \rightarrow \psi := \neg\varphi \vee \psi$, $\top := (p \vee \neg p)$ for some atomic predicate p, $\diamondsuit\!\!\!-\varphi := \top S \varphi$ ("once φ"), $\boxminus \varphi := \neg \diamondsuit\!\!\!- \neg\varphi$ ("so far φ"), $\Diamond\varphi := \top U \varphi$ ("eventually φ"), $\Box\varphi := \neg \Diamond \neg\varphi$ ("always φ"), $\varphi W \psi := \varphi U \psi \vee \Box\varphi$ ("weak until"), $\exists x(\varphi) := \neg\forall x(\neg\varphi)$.

4 Sequential Consistency

We present a trace-based definition of sequential consistency (cf., [1]) and prove our logical characterization equivalent. Our definition of sequential consistency generalizes the original definition [16] by allowing non-sequential specifications.

Definition 1 (Sequential Consistency). *Let* $\textsc{Spec} \subseteq \mathcal{E}^*$ *be a specification of the shared data-structure. A trace* $E \downharpoonright i$ *is sequentially consistent* $seqCons(E, i)$ *if and only if there is* $(E', i') \in \mathcal{E}^\omega \times \mathbb{N}$ *s.t. for all* $t \in \textsc{Threads}$*:*

$$(E \downharpoonright i) \downarrow t = (E' \downharpoonright i') \downarrow t \text{ and } E' \downharpoonright i' \in \textsc{Spec} .$$

Basic Predicates. For our logical characterization, we define the predicate *correct* representing the fact that a trace meets the specification:

$$(E, i) \vDash \mathit{correct} \text{ :iff } E \downharpoonright i \in \textsc{Spec} .$$

Theorem 1 (Logical Characterization of Sequential Consistency). *A trace* $E \downharpoonright i$ *is sequentially consistent if and only if the threads do not jointly know that it is incorrect:* $seqCons(E, i)$ *iff* $(E, i) \vDash \neg D_{\textsc{Threads}}(\neg \mathit{correct})$.

5 Eventual Consistency

We define the set of actions for eventual consistency as:

$$\mathcal{A} \ni act := \mathit{qu}(id, q, r) \mid \mathit{up}(id, u) \mid \mathit{com}(id) \mid \mathit{fwd}(t, t', id) .$$

Threads may pose a query (qu) $q \in \textsc{Queries}$ with result $r \in \textsc{Values}$, issue an update (up) $u \in \textsc{Updates}$, or commit (com) their local changes. Queries, updates and commits get assigned a revision-id $id \in \textsc{Identifiers}$, representing the current state of the local database copy. We assume that if a thread commits, the committed revision id matches the revision id of the previous queries and updates i.e., those performed since the last commit, and that thread-revision-id pairs (t, id) are unique. Again, this is no restriction. To fulfil the requirement, the threads can just increment their local revision id whenever they commit. As updates may get lost in the network, we represent by $\mathit{fwd}(t, t', id)$ the successful forwarding of the updates made by thread t in revision id to thread t'.

Preliminaries. We let $set(E) = \{e \mid e \in E\}$, i.e., the set of events in trace E. On a fixed trace E, we define the program order $\prec_p$ as $e \prec_p e'$:iff if there is t such that $pos(e, E \downarrow t) < pos(e', E \downarrow t)$. We let "$\_$" represent irrelevant, existential quantification. Let $e \equiv_t e'$ if and only if there is $id \in \textsc{Identifiers}$ such that $e = (t, \_(id, \_))$ and $e' = (t, \_(id, \_))$, i.e., if the events belong to the same revision of thread t. A relation $\preceq$ factors over $\equiv_t$ if $x \preceq y$, $x \equiv_t x'$ and $y \equiv_t y'$ imply $x' \preceq y'$. Updates are interpreted in terms of a set of states $\textsc{States}$, i.e., we assume there is an interpretation function $u^\# : \textsc{States} \to \textsc{States}$, for each $u \in \textsc{Updates}$, and a designated initial state $s_0 \in \textsc{States}$. For each query $q \in \textsc{Queries}$, there is an interpretation function $q^\# : \textsc{States} \to \textsc{Values}$. For a finite set of events E_S, a total order $\prec$ over the events in E_S, and a state s we let $\mathit{apply}(E_s, \prec, s)$ be the result of applying all updates in E_s to s, in the order specified by $\prec$.

Definition 2 (Eventual Consistency). *We use the definition presented in [5]. A trace $E \in \mathcal{E}^\omega$ is eventually consistent ($evCons(E)$) if and only if there exist a partial order $\prec_v$ (visibility order), and a total order $\prec_a$ (arbitration order) on the events in $set(E)$ such that:*

- $\prec_v \subseteq \prec_a$ *(arbitration extends visibility).*
- $\prec_p \subseteq \prec_v$ *(visibility is compatible with program-order).*
- *for each $e_q = (t, qu(id, q, r)) \in E$, we have $r = q^\#(apply(\{e \mid e \prec_v e_q\}, \prec_a, s_0))$ (consistent query results).*
- $\prec_a$ *and* $\prec_v$ *factor over* $\equiv_t$ *(atomic revisions).*
- *if $(t, com(id)) \notin E$ and $(t, \_(id, \_)) \prec_v (t', \_)$ then $t = t'$ (uncommitted updates).*
- *if $e = (t, com(id)) \in E$ then there are only finitely many $e' := (t', com(id'))$ such that $e' \in E$ and $e \not\prec_v e'$ (eventual visibility).*

5.1 Logical Characterization

Basic Predicates. We represent queries and updates by predicates $query(t, q, r, id)$ and $update(t, u, id)$, representing $t \in$ Threads, issuing $q \in$ Queries with result $r \in$ Values on revision $id \in$ Identifiers, and t performing $u \in$ Updates on revision id, respectively. As threads work on their local copies, revision ids mark the version of data the threads work with. We represent commits by the predicate $commit(t, id)$, representing t committing its state in revision id. After performing a commit, a new revision id is assigned. We define:

$$\begin{aligned}(E, i) &\vDash query(t, q, r, id) &&\text{:iff } E@i = (t, qu(id, q, r))\\ (E, i) &\vDash update(t, u, id) &&\text{:iff } E@i = (t, up(id, u))\\ (E, i) &\vDash commit(t, id) &&\text{:iff } E@i = (t, com(id))\ .\end{aligned}$$

We let $query(t, q, r) := \exists id(query(t, q, r, id))$. Upon commit, a thread forwards all the information from its local repository to the database system and receives updates from other threads. Committed updates may however be delayed or lost by the network. By the predicate $forward(t, t', id)$ we mark the event that the environment forwarded the updates t performed in revision id to t'. We let: $(E, i) \vDash forward(t, t', id)$:iff $E@i = (env, fwd(t, t', id))$. Eventual consistency requires all threads to keep valid logs. Logs are finite sequences of actions, i.e., $\mathcal{L} \in A^*$. We let: $(E, i) \vDash a \text{ in } \mathcal{L}$:iff $a \in L$. By $t\ k_{log}\ a$ we denote the fact that t knows about action a. The predicate k_{log} represents individual knowledge, i.e., knowledge in the sense of knowing about an action in contrast to knowing that a fact is true [15]. We let:

$$\begin{aligned}(E, i) \vDash t\ k_{log}\ a \text{ :iff there is } j \le i : (E@j = (t, a) \text{ or}\\ ((E, j) \vDash forward(t', t, id) \text{ and there is } l < j : (E, l) \vDash commit(t', id)\\ \text{and } (E, l) \vDash t'\ k_{log}\ a))\ .\end{aligned}$$

That is, threads know an action if they performed it themselves, or they received an update containing it. Upon commits, threads pass on all actions they know

about. We represent log validity by the formula: $\mathcal{L}\ validLog\ t\ :=\ \forall a(t\ k_{log}\ a \leftrightarrow a\ in\ \mathcal{L}) \wedge consistent(\mathcal{L})$. That is, to be a valid log for thread t, log $\mathcal{L}$ must contain exactly the actions that t knows of and these actions must be arranged in an order consistent with respect to the other threads logs. A log $\mathcal{L}$ is consistent if the actions in the log occur in the same order as the actions in the real trace. This means the sequence of actions in $\mathcal{L}$ must be a subsequence of the actions in the real trace. A sequence $a = a_1 a_2 \dots a_n$ is a subsequence of a sequence $b = b_1 b_2 \dots b_m$ ($a \preceq b$), if and only if there exist $1 \leq i_1 < i_2 < \dots < i_n \leq m$ such that for all $1 \leq j \leq n : a_j = b_{i_j}$. We project a sequence of events to a sequence of actions by the function $act : \mathcal{E}^* \to \mathcal{A}^*$, such that $act((t_1, a_1)(t_2, a_2) \dots (t_n, a_n)) = a_1 a_2 \dots a_n$. We define: $(E, i) \vDash consistent(\mathcal{L})$:iff $\mathcal{L} \preceq act(E \downarrow i)$.

Query Results. All queries that threads issue must return the correct result with respect to the logged operations. That is, the query's result must match the result the query would yield when issued on a database that performed all the updates in the log. We represent the fact that query q would yield result r on log $\mathcal{L}$ by the predicate $\text{result}(q, \mathcal{L}, r)$. We define the order of actions in a log $\mathcal{L}$ by the relation $<_{\mathcal{L}}$. We let $a <_{\mathcal{L}} a'$:iff $pos(a, \mathcal{L}) < pos(a', \mathcal{L}) < \omega$. We define: $(E, i) \vDash result(q, \mathcal{L}, r)$:iff $r = q^{\#}(apply(set(\mathcal{L}), <_{\mathcal{L}}, s_0))$.

Network Assumptions. We pose additional requirements on the network: updates in the same revision must be sent as atomic bundles (*atomicTrans*). Only committed updates can be forwarded (*fwd*). Active threads must eventually receive all committed update (*alive*). We define the helper predicate: $rev(t, id) := \exists q \exists r(query(t, q, r, id)) \vee \exists u(update(t, u, id))$ representing the fact, that the current action belongs to revision id of thread t. We specify the requirements that updates made in the same revision must be sent bundled as indivisible transactions by the formula : $atomicTrans := \forall t \forall id(\boxminus(rev(t, id) \rightarrow rev(t, id)\ W\ commit(t, id)))$. That is, queries and updates from revision id are only followed by other queries and updates from the same revision, or a commit. We enforce that only committed revisions can be forwarded by: $fwd := \forall t \forall t' \forall id(\boxminus(fwd(t, t', id) \rightarrow \Box(\neg commit(t, id))))$. Threads that makes progress, i.e. that commit infinitely often must eventually receive all committed updates. We formalize this as:

$$
\begin{aligned}
alive :=\ & \forall t \forall t' \forall id \\
& (\boxminus(commit(t, id) \wedge \Box \Diamond (\exists id'(commit(t', id'))) \rightarrow \\
& \Diamond forward(t, t', id))) .
\end{aligned}
$$

We represent *correctEVC* by the formula:

$$
\begin{aligned}
correctEVC :=\ & \forall t \forall q \forall r \\
& (\boxminus(query(t, q, r) \rightarrow \exists \mathcal{L}(\mathcal{L}\ validLog\ t\ \wedge result(q, \mathcal{L}, r)))) \\
& \wedge atomicTrans \wedge alive \wedge fwd\ .
\end{aligned}
$$

Theorem 2 (Logical Characterization of Eventual Consistency). *A trace is eventually consistent if and only if the threads do not know that it violates correctEVC. For all traces* $E \in \mathcal{E}^\omega$*:*

$$evCons(E) \text{ if and only if } E \vDash \neg D_{\text{THREADS}} \neg (correctEVC) \ .$$

6 Linearizability

Linearizability refines sequential consistency by guaranteeing that each method call takes its effect at exactly one point between its invocation and its return.

For our definition of linearizability, we follow [8]. As for sequential consistency, our definition generalizes the original notion [10, 12] by allowing non-sequential specifications. We define the real-time precedence order $\preceq_{real} \subseteq (\mathcal{E}^\omega \times \mathbb{N})^2$: $(E, i) \preceq_{real} (E', i')$:iff $i = i'$ and there is a bijection $\pi : \{1, \ldots, i\} \to \{1, \ldots, i\}$ s.t for all $j \in \mathbb{N}$ such that $j \leq i : E@j = E'@\pi(j)$, i.e., E' is a permutation of E, and for all $j, k \in \mathbb{N}$ such that $j < k \leq i :$ if $E@j \in \text{RET}$ and $E@k \in \text{INV}$ then $\pi(j) < \pi(k)$, i.e., when permuting the events in E, calls are never pulled before returns.

Definition 3 (Linearizability). *A trace* (E, i) *is linearizable (*$lin(E, i)$*) if and only if there is* $(E', i') \in \mathcal{E}^\omega \times \mathbb{N}$ *such that (1) for all* $t \in \text{THREADS}$*:* $(E \downharpoonright i) \downarrow t = (E' \downharpoonright i') \downarrow t$ *(2)* $(E, i) \preceq_{real} (E', i')$ *and (3)* $E' \downharpoonright i' \in \text{SPEC}$*.*

Theorem 3 (Logical Characterization of Linearizability). *A trace* $E \downharpoonright i \in \mathcal{E}^*$ *is linearizable if and only if the threads together with the observer do not know that it is incorrect:*

$$lin(E, i) \text{ iff } (E, i) \vDash \neg D_{\text{THREADS} \uplus \{obs\}} \neg correct \ .$$

7 Knowledge about Consistency

We write $\vDash \varphi$ as an abbreviation for: for all $E \in \mathcal{E}^\omega$: $E \vDash \varphi$. Let $seqCons := \neg D_{\text{THREADS}}(\neg correct)$.

Theorem 4 (Detection Sequential Consistency). *Threads can decide whether a trace is sequentially consistent or not:* $\vDash (seqCons \leftrightarrow D_{Threads}(seqCons)) \wedge (\neg seqCons \leftrightarrow D_{Threads}(\neg seqCons))$.

Let $Lin := \neg D_{\text{THREADS} \uplus \{obs\}} \neg correct$.

Theorem 5 (Detection Linearizability). *There is* $E \in \mathcal{E}^\omega$ *such that* $(E, i) \vDash Lin \wedge \neg D_{Threads \uplus \{obs\}}(Lin)$*. As in sequential consistency, the threads together with the observer can spot if a trace is not linearizable:* $\vDash \neg Lin \leftrightarrow D_{Threads \uplus \{obs\}}(\neg Lin)$.

8 Related Work

The only applications of epistemic logic to concurrent computations that we are aware of are a logical characterization of wait-free computations by Hirai [13] and a knowledge based analysis of cache-coherence by Baukus et al. [2].

Acknowledgements. We would like to thank Jade Alglave, Alexey Gotsman, Rose Hoberman, Simon Kramer, Corneliu Popeea, Moshe Y. Vardi, and the anonymous reviewers for helpful feedback. This research was supported in part by a Microsoft Research Scholarship and by the ERC project 308125 VeriSynth.

References

1. Attiya, H., Welch, J.L.: Sequential consistency versus linearizability. ACM Trans. Comput. Syst. 12(2), 91–122 (1994)
2. Baukus, K., van der Meyden, R.: A knowledge based analysis of cache coherence. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 99–114. Springer, Heidelberg (2004)
3. Belardinelli, F., Lomuscio, A.: A complete first-order logic of knowledge and time. In: KR (2008)
4. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: PLDI (2010)
5. Burckhardt, S., Leijen, D., Fähndrich, M., Sagiv, M.: Eventually consistent transactions. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 67–86. Springer, Heidelberg (2012)
6. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: POPL (2009)
7. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press (2003)
8. Gotsman, A., Yang, H.: Liveness-preserving atomicity abstraction. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 453–465. Springer, Heidelberg (2011)
9. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. JACM 37(3), 549–587 (1990)
10. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM TOPLAS 12(3), 463–492 (1990)
11. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2008)
12. Herlihy, M.P., Wing, J.M.: Axioms for concurrent objects. In: POPL (1987)
13. Hirai, Y.: An intuitionistic epistemic logic for sequential consistency on shared memory. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 272–289. Springer, Heidelberg (2010)
14. Hoare, C.A.R.: An axiomatic basis for computer programming. CACM 12(10), 576–580 (1969)
15. Kramer, S., Rybalchenko, A.: A multi-modal framework for achieving accountability in multi-agent systems. In: LIS (2010)
16. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers C-28(9), 690–691 (1979)
17. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
18. Papadimitriou, C.H.: The serializability of concurrent database updates. JACM 26(4), 631–653 (1979)
19. Shapiro, M., Kemme, B.: Eventual consistency. In: Özsu, M.T., Liu, L. (eds.) Encyclopedia of Database Systems, pp. 1071–1072. Springer, Heidelberg (2009)
20. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010)
21. van Ditmarsch, H., van der Hoeck, W., Kooi, B.: Dynamic Epistemic Logic. Springer, Dordrecht (2008)

Characterizing Progress Properties of Concurrent Objects via Contextual Refinements

Hongjin Liang[1,2], Jan Hoffmann[2], Xinyu Feng[1], and Zhong Shao[2]

[1] University of Science and Technology of China
[2] Yale University

Abstract. Implementations of concurrent objects should guarantee linearizability and a progress property such as wait-freedom, lock-freedom, obstruction-freedom, starvation-freedom, or deadlock-freedom. Conventional informal or semi-formal definitions of these progress properties describe conditions under which a method call is guaranteed to complete, but it is unclear how these definitions can be utilized to formally verify system software in a layered and modular way.

In this paper, we propose a unified framework based on contextual refinements to show exactly how progress properties affect the behaviors of client programs. We give formal operational definitions of all common progress properties and prove that for linearizable objects, each progress property is equivalent to a specific type of contextual refinement that preserves termination. The equivalence ensures that verification of such a contextual refinement for a concurrent object guarantees both linearizability and the corresponding progress property. Contextual refinement also enables us to verify safety and liveness properties of client programs at a high abstraction level by soundly replacing concrete method implementations with abstract atomic operations.

1 Introduction

A concurrent object consists of shared data and a set of methods that provide an interface for client threads to manipulate and access the shared data. The synchronization of simultaneous data access within the object affects the progress of the execution of the client threads in the system.

Various progress properties have been proposed for concurrent objects. The most important ones are wait-freedom, lock-freedom and obstruction-freedom for non-blocking implementations, and starvation-freedom and deadlock-freedom for lock-based implementations. These properties describe conditions under which method calls are guaranteed to successfully complete in an execution. For example, lock-freedom guarantees that "infinitely often some method call finishes in a finite number of steps" [9].

Nevertheless, the common informal or semi-formal definitions of the progress properties are difficult to use in a modular and layered program verification because they fail to describe how the progress properties affect clients. In a modular

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 227–241, 2013.

verification of client threads, the concrete implementation Π of the object methods should be replaced by an abstraction (or specification) Π_A that consists of equivalent atomic methods. The progress properties should then characterize whether and how the behaviors of a client program will be affected if a client uses Π instead of Π_A. In particular, we are interested in systematically studying whether the termination of a client using the abstract methods Π_A will be preserved when using an implementation Π with some progress guarantee.

Previous work on verifying the *safety* of concurrent objects (*e.g.*, [4,12]) has shown that linearizability—a standard safety criterion for concurrent objects—and contextual refinement are equivalent. Informally, an implementation Π is a contextual refinement of a (more abstract) implementation Π_A, if every observable behavior of any client program using Π can also be observed when the client uses Π_A instead. To obtain equivalence to linearizability, the observable behaviors include I/O events but not divergence (*i.e.*, non-termination). Recently, Gotsman and Yang [6] showed that a client program that diverges using a linearizable and *lock-free* object must also diverge when using the abstract operations instead. Their work reveals a connection between lock-freedom and a form of contextual refinement which preserves termination as well as safety properties. It is unclear how other progress guarantees affect termination of client programs and how they are related to contextual refinements.

This paper studies all five commonly used progress properties and their relationships to contextual refinements. We propose a unified framework in which a certain type of termination-sensitive contextual refinement is equivalent to linearizability together with one of the progress properties. The idea is to identify different observable behaviors for different progress properties. For example, for the contextual refinement for lock-freedom we observe the divergence of the whole program, while for wait-freedom we also need to observe which threads in the program diverge. For lock-based progress properties, *e.g.*, starvation-freedom and deadlock-freedom, we have to take fair schedulers into account.

Our paper makes the following new contributions:

- We formalize the definitions of the five most common progress properties: wait-freedom, lock-freedom, obstruction-freedom, starvation-freedom, and deadlock-freedom. Our formulation is based on possibly infinite event traces that are operationally generated by any client using the object.
- Based on our formalization, we prove relationships between the progress properties. For example, wait-freedom implies lock-freedom and starvation-freedom implies deadlock-freedom. These relationships form a lattice shown in Figure 1 (where the arrows represent implications). We close the lattice with a bottom element that we call *sequential termination*, a progress property in the sequential setting. It is weaker than any other progress property.
- We develop a unified framework to characterize progress properties via contextual refinements. With linearizability, each progress property is proved equivalent to a contextual refinement which takes into account divergence of programs. A companion TR [14] contains the formal proofs of our results.

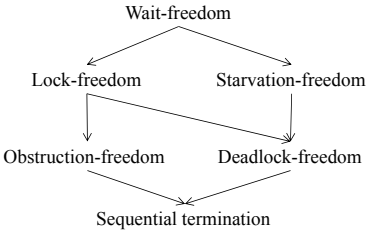


Fig. 1. Relationships between Progress Properties

By extending earlier equivalence results on linearizability [4], our contextual refinement framework can serve as a new alternative definition for the full correctness properties of concurrent objects. The contextual refinement implied by linearizability and a progress guarantee precisely characterizes the properties at the abstract level that are preserved by the object implementation. When proving these properties of a client of the object, we can soundly replace the concrete method implementations by its abstract operations. On the other hand, since the contextual refinement also implies linearizability and the progress property, we can potentially borrow ideas from existing proof methods for contextual refinements, such as simulations (*e.g.*, [13]) and logical relations (*e.g.*, [2]), to verify linearizability and the progress guarantee together.

In the remainder of this paper, we first informally explain our framework in Section 2. We then introduce the formal setting in Section 3; including the definition of linearizability as the safety criterion of objects. We formulate the progress properties in Section 4 and the contextual refinement framework in Section 5. We discuss related work and conclude in Section 6.

2 Informal Account

In this section, we informally describe our results. We first give an overview of linearizability and its equivalence to the basic contextual refinement. Then we explain the progress properties and summarize our new equivalence results.

Linearizability and Contextual Refinement. *Linearizability* is a standard safety criterion for concurrent objects [9]. Intuitively, linearizability describes atomic behaviors of object implementations. It requires that each method call should appear to take effect instantaneously at some moment between its invocation and return.

Linearizability intuitively establishes a correspondence between the object implementation Π and the intended atomic operations Π_A. This correspondence can also be understood as a *contextual refinement*. Informally, we say that Π is a contextual refinement of Π_A, $\Pi \sqsubseteq \Pi_A$, if substituting Π for Π_A in any context (*i.e.*, in a client program) does not add observable behaviors. External observers cannot tell that Π_A has been replaced by Π from monitoring the behaviors of the client program.

It has been proved [4,12] that linearizability is equivalent to a contextual refinement in which the observable behaviors are finite traces of I/O events. Thus this basic contextual refinement can be used to distinguish linearizable objects from non-linearizable ones. But it cannot characterize progress properties of objects.

Progress Properties. Figure 2 shows several implementations of a counter with different progress guarantees that we study in this paper. A counter object provides the two methods `inc` and `dec` for incrementing and decrementing a shared variable `x`. The implementations given here are not intended to be practical but merely to demonstrate the meanings of the progress properties. We assume that every command is executed atomically.

Informally, an object implementation is *wait-free*, if it guarantees that every thread can complete any started operation of the data structure in a finite number of steps [7]. Figure 2(a) shows an ideal wait-free implementation in which the increment and the decrement are done atomically. This implementation is obviously wait-free since it guarantees termination of every method call regardless of interference from other threads. Note that realistic implementations of wait-free counters are more complex and involve arrays and atomic snapshots [1].

Lock-freedom is similar to wait-freedom but only guarantees that *some* thread will complete an operation in a finite number of steps [7]. Typical lock-free implementations (such as the well-known Treiber stack, HSY elimination-backoff stack and Harris-Michael lock-free list) use the atomic compare-and-swap instruction `cas` in a loop to repeatedly attempt an update until it succeeds. Figure 2(b) shows such an implementation of the counter object. It is lock-free, because whenever `inc` and `dec` operations are executed concurrently, there always exists some successful update. Note that this object is not wait-free. For the following program (2.1), the `cas` instruction in the method called by the left thread may continuously fail due to the continuous updates of `x` made by the right thread.

```
inc();   ||   while(true) inc();                    (2.1)
```

Herlihy *et al.* [8] propose *obstruction-freedom* which "guarantees progress for any thread that eventually executes in isolation" (*i.e.*, without other active threads in the system). They present two double-ended queues as examples. In Figure 2(c) we show an obstruction-free counter that may look contrived but nevertheless illustrates the idea of the progress property.

The implementation introduces a variable `i`, and lets `inc` perform the atomic increment after increasing `i` to 10 and `dec` do the atomic decrement after decreasing `i` to 0. Whenever a method is executed in isolation (*i.e.*, without interference from other threads), it will complete. Thus the object is obstruction-free. It is not lock-free, because for the client

```
inc();   ||   dec();                               (2.2)
```

which executes an increment and a decrement concurrently, it is possible that neither of the method calls returns. For instance, under a specific schedule, every increment over `i` made by the left thread is immediately followed by a decrement from the right thread.

```
1 inc() { x := x + 1; }
2 dec() { x := x - 1; }
```

(a) Wait-Free (Ideal) Impl.

```
1 inc() {
2  local t, b;
3  do {
4    t := x;
5    b := cas(&x,t,t+1);
6  } while(!b);
7 }
```

(b) Lock-Free Impl.

```
 1 inc() {
 2   while (i < 10) {
 3     i := i + 1;
 4   }
 5   x := x + 1;
 6 }
 7 dec() {
 8   while (i > 0) {
 9     i := i - 1;
10   }
11   x := x - 1;
12 }
```

(c) Obstruction-Free Impl.

```
1 inc() {
2   TestAndSet_lock();
3   x := x + 1;
4   TestAndSet_unlock();
5 }
```

(d) Deadlock-Free Impl.

```
 1  inc() {
 2    Bakery_lock();
 3    x := x + 1;
 4    Bakery_unlock();
 5  }
```

(e) Starvation-Free Impl.

Fig. 2. Counter Objects with Methods `inc` and `dec`

Wait-freedom, lock-freedom, and obstruction-freedom are progress properties for non-blocking implementations, where a delay of a thread cannot prevent other threads from making progress. In contrast, deadlock-freedom and starvation-freedom are progress properties for lock-based implementations. A delay of a thread holding a lock will block other threads which request the lock.

Deadlock-freedom and starvation-freedom are often defined in terms of locks and critical sections. Deadlock-freedom guarantees that some thread will succeed in acquiring the lock, and starvation-freedom states that every thread attempting to acquire the lock will eventually succeed [9]. For example, a test-and-set spin lock is deadlock-free but not starvation-free. In a concurrent access, some thread will successfully set the bit and get the lock, but there might be a thread that is continuously failing to get the lock. Lamport's bakery lock is starvation-free. It ensures that threads can acquire locks in the order of their requests.

However, as noted by Herlihy and Shavit [10], the above definitions based on locks are unsatisfactory, because it is often difficult to identify a particular field in the object as a lock. Instead, they suggest defining them in terms of method calls. They also notice that the above definitions implicitly assume that every thread acquiring the lock will eventually release it. This assumption requires *fair* scheduling, *i.e.*, every thread gets eventually executed.

Following Herlihy and Shavit [10], we say an object is *deadlock-free*, if in each *fair* execution there always exists some method call that can finish. As an example in Figure 2(d), we use a test-and-set lock to synchronize the increments of the counter. Since some thread is guaranteed to acquire the test-and-set lock, the method call of that thread is guaranteed to finish. Thus the object is deadlock-free. Similarly, a *starvation-free* object guarantees that every method call can finish in fair executions. Figure 2(e) shows a counter implemented with Lamport's bakery lock. It is starvation-free since the bakery lock ensures that every thread can acquire the lock and hence every method call can eventually complete.

Table 1. Characterizing Progress Properties via Contextual Refinements $\Pi \sqsubseteq \Pi_A$

	Wait-Free	Lock-Free	Obstruction-Free	Deadlock-Free	Starvation-Free
Π_A	(t, Div.)	Div.	Div.	Div.	(t, Div.)
Π	(t, Div.)	Div.	Div. if Isolating	Div. if Fair	(t, Div.) if Fair

Our Results. None of the above definitions of the five progress properties describes their guarantees regarding the behaviors of client code. In this paper, we define several contextual refinements to characterize the effects over client behaviors when the client uses objects with some progress properties. We show that linearizability together with a progress property is equivalent to a certain termination-sensitive contextual refinement. Table 1 summarizes our results.

For each progress property, the new contextual refinement $\Pi \sqsubseteq \Pi_A$ is defined with respect to a divergence behavior and/or a specific scheduling at the implementation level (the third row in Table 1) and at the abstract side (the second row), in addition to the I/O events in the basic contextual refinement for linearizability.

- For wait-freedom, we need to observe the divergence of each individual thread t, represented by "(t, Div.)" in Table 1, at both the concrete and the abstract levels. We show that, if the thread t of a client program diverges when the client uses a linearizable and wait-free object Π, then thread t must also diverge when using Π_A instead.
- The case for lock-freedom is similar, except that we now consider the divergence behaviors of the *whole* client program rather than individual threads (denoted by "Div." in Table 1). If a client diverges when using a linearizable and lock-free object Π, it must also diverge when it uses Π_A instead.
- For obstruction-freedom, we consider the behaviors of *isolating* executions at the concrete side (denoted by "Div. if Isolating" in Table 1). In those executions, eventually only one thread is running. We show that, if a client diverges in an isolating execution when it uses a linearizable and obstruction-free object Π, it must also diverge in some abstract execution.
- For deadlock-freedom, we only care about *fair* executions at the concrete level (denoted by "Div. if Fair" in Table 1).
- For starvation-freedom, we observe the divergence of each individual thread at both levels and restrict our considerations to fair executions for the concrete side ("(t, Div.) if Fair" in Table 1). Any thread using Π can diverge in a fair execution, only if it also diverges in some abstract execution.

These new contextual refinements can characterize linearizable objects with progress properties. We will formalize the results and give examples in Section 5.

3 Formal Setting and Linearizability

In this section, we formalize linearizability and show its equivalence to a contextual refinement that preserves safety properties only. This equivalence is the basis of our new results that relate progress properties and contextual refinements.

$$
\begin{array}{rrcl}
(\mathit{Expr}) & E & ::= & \ldots \quad (\mathit{BExp})\ B ::= \ldots \quad (\mathit{Instr})\ c ::= \mathbf{print}(E) \mid \ldots \\
(\mathit{Stmt}) & C & ::= & \mathbf{skip} \mid c \mid x := f(E) \mid \mathbf{return}\ E \mid \mathbf{end} \\
 & & & \mid \langle C \rangle \mid C; C \mid \mathbf{if}\ (B)\ C\ \mathbf{else}\ C \mid \mathbf{while}\ (B)\{C\} \\
(\mathit{Prog}) & W & ::= & \mathbf{skip} \mid \mathbf{let}\ \Pi\ \mathbf{in}\ C \| \ldots \| C \\
(\mathit{ODecl}) & \Pi & ::= & \{f_1 \rightsquigarrow (x_1, C_1), \ldots, f_n \rightsquigarrow (x_n, C_n)\}
\end{array}
$$

Fig. 3. Syntax of the Programming Language

$$
\begin{array}{rrcl}
(\mathit{State}) & \mathcal{S} & ::= & \ldots \qquad (\mathit{ThrdID})\ \mathsf{t} \in \mathit{Nat} \\
(\mathit{Evt}) & e & ::= & (\mathsf{t}, f, n) \mid (\mathsf{t}, \mathbf{ret}, n) \mid (\mathsf{t}, \mathbf{obj}) \mid (\mathsf{t}, \mathbf{obj}, \mathbf{abort}) \\
 & & & \mid (\mathsf{t}, \mathbf{out}, n) \mid (\mathsf{t}, \mathbf{clt}) \mid (\mathsf{t}, \mathbf{clt}, \mathbf{abort}) \mid (\mathsf{t}, \mathbf{term}) \mid (\mathbf{spawn}, n) \\
(\mathit{ETrace}) & T & ::= & \epsilon \mid e :: T \quad \text{(co-inductive)}
\end{array}
$$

Fig. 4. States and Event Traces

Language and Semantics. We use a similar language as in previous work of Liang and Feng [12]. As shown in Figure 3, a program W consists of several client threads that run in parallel. Each thread could call the methods declared in the object Π. A method f is defined as a pair (x, C), where x is the formal argument and C is the method body. The object Π could be either concrete with fine-grained code that we want to verify, or abstract (usually denoted as Π_A in the following) that we consider as the specification. For the latter case, each method body should be an atomic operation of the form $\langle C \rangle$ and it should be always safe to execute it. For simplicity, we assume there is only one object in the program W and each method takes one argument only.

Most commands are standard. Clients can use $\mathbf{print}(E)$ to produce observable external events. We do not allow the object's methods to produce external events. To simplify the semantics, we also assume there are no nested method calls. To discuss progress properties later, we introduce an auxiliary command **end**. It is a special marker that can be added at the end of a thread, but is not supposed to be used directly by programmers. The **skip** statement plays two roles here: a statement that has no computation effects or a flag to show the end of an execution.

We use $\mathcal{S}$ for a program state. Program transitions $(W, \mathcal{S}) \overset{e}{\longmapsto} (W', \mathcal{S}')$ generate events e defined in Figure 4. A method invocation event (t, f, n) is produced when thread t executes $x := f(E)$, where n is the value of the argument E. A return $(\mathsf{t}, \mathbf{ret}, n)$ is produced with the return value n. $\mathbf{print}(E)$ generates an output $(\mathsf{t}, \mathbf{out}, n)$, and **end** generates a termination marker $(\mathsf{t}, \mathbf{term})$. Other steps generate either normal object actions $(\mathsf{t}, \mathbf{obj})$ (for steps inside method calls) or silent client actions $(\mathsf{t}, \mathbf{clt})$ (for client steps other than $\mathbf{print}(E)$). For transitions leading to the error state **abort** (*e.g.*, invalid memory access), fault events are produced: $(\mathsf{t}, \mathbf{obj}, \mathbf{abort})$ by the object method code and $(\mathsf{t}, \mathbf{clt}, \mathbf{abort})$ by the client code. We also introduce an auxiliary event $(\mathbf{spawn}, n)$, saying that n threads are spawned. It will be useful later when defining fair scheduling (in Section 4). We write $\mathsf{tid}(e)$ for the thread ID in the event e. The predicate $\mathsf{is\_clt}(e)$

$$\mathcal{T}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{T \mid \exists W', \mathcal{S}'.\ (W,\mathcal{S}) \stackrel{T}{\longmapsto}{}^{*} (W',\mathcal{S}') \vee (W,\mathcal{S}) \stackrel{T}{\longmapsto}{}^{*} \mathbf{abort}\}$$
$$\mathcal{H}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_hist}(T) \mid T \in \mathcal{T}[\![W,\mathcal{S}]\!]\}$$
$$\mathcal{O}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}[\![W,\mathcal{S}]\!]\}$$

Fig. 5. Generation of Finite Event Traces

states that the event e is either a silent client action, an output, or a client fault. We write $\mathsf{is\_inv}(e)$ and $\mathsf{is\_ret}(e)$ to denote that e is a method invocation and a return, respectively. The predicate $\mathsf{is\_abt}(e)$ denotes a fault of the object or the client. Method invocations, returns and object faults are called *history* events, which will be used to define linearizability below. Outputs, client faults and object faults are called *observable* events.

An event trace T is a finite or infinite sequence of events. We write $T(i)$ for the i-th event of T. $\mathsf{last}(T)$ is the last event in a finite T. The trace $T(1..i)$ is the sub-trace $T(1), \ldots, T(i)$ of T, and $|T|$ is the length of T ($|T| = \omega$ if T is infinite). The trace $T|_{\mathsf{t}}$ represents the sub-trace of T consisting of all events whose thread ID is t. We can use $\mathsf{get\_hist}(T)$ to project T to the sub-trace consisting of all the history events, and $\mathsf{get\_obsv}(T)$ for the sub-trace of all the observable events. Finite traces of history events are called *histories*.

In Figure 5, we define $\mathcal{T}[\![W,\mathcal{S}]\!]$ for the prefix-closed set of finite traces produced by the executions of $(W,\mathcal{S})$. We use $(W,\mathcal{S}) \stackrel{T}{\longmapsto}{}^{*} (W',\mathcal{S}')$ for zero or multiple-step program transitions that generate the trace T. We also define $\mathcal{H}[\![W,\mathcal{S}]\!]$ and $\mathcal{O}[\![W,\mathcal{S}]\!]$ to get histories and finite observable traces produced by the executions of $(W,\mathcal{S})$. The TR [14] contains more details about the language.

Linearizability and Basic Contextual Refinement. We formulate linearizability following its standard definition [11]. Below we sketch the basic concepts. Detailed formal definitions can be found in the companion TR [14].

Linearizability is defined using histories. We say a return e_2 *matches* an invocation e_1, denoted as $\mathsf{match}(e_1, e_2)$, iff they have the same thread ID. An invocation is *pending* in T if no matching return follows it. We can use $\mathsf{pend\_inv}(T)$ to get the set of pending invocations in T. We handle pending invocations in a history T in the standard way [11]: we append zero or more return events to T, and drop the remaining pending invocations. The result is denoted by $\mathsf{completions}(T)$. It is a set of histories, and for each history in it, every invocation has a matching return event.

Definition 1 (Linearizable Histories). $T \preceq_{\mathsf{lin}} T'$ *iff*

1. $\forall \mathsf{t}.\ T|_{\mathsf{t}} = T'|_{\mathsf{t}}$;
2. *there exists a bijection* $\pi : \{1, \ldots, |T|\} \to \{1, \ldots, |T'|\}$ *such that* $\forall i.\ T(i) = T'(\pi(i))$ *and* $\forall i, j.\ i < j \wedge \mathsf{is\_ret}(T(i)) \wedge \mathsf{is\_inv}(T(j)) \implies \pi(i) < \pi(j)$.

That is, T is linearizable *w.r.t.* T' if the latter is a permutation of the former, preserving the order of events in the same threads and the order of the non-overlapping method calls. Then an *object* is linearizable iff each of its concurrent histories after completions is linearizable *w.r.t.* some *legal sequential* history.

We use $\Pi_A \rhd (\mathcal{S}_a, T')$ to mean that T' is a legal sequential history generated by any client using the specification Π_A with an abstract initial state $\mathcal{S}_a$.

Definition 2 (Linearizability of Objects). *The object's implementation Π is linearizable w.r.t. Π_A under a refinement mapping φ, denoted by $\Pi \preceq_\varphi \Pi_A$, iff*

$$\forall n, C_1, \ldots, C_n, \mathcal{S}, \mathcal{S}_a, T.\ T \in \mathcal{H}[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), \mathcal{S}]\!] \wedge (\varphi(\mathcal{S}) = \mathcal{S}_a)$$
$$\Longrightarrow \exists T_c, T'.\ T_c \in \mathsf{completions}(T) \wedge \Pi_A \rhd (\mathcal{S}_a, T') \wedge T_c \preceq_{\mathsf{lin}} T'\,.$$

Here the partial mapping φ: State $\rightharpoonup$ State relates concrete states to abstract ones.

The side condition $\varphi(\mathcal{S}) = \mathcal{S}_a$ in the above definition requires the initial concrete state $\mathcal{S}$ to be well-formed in that it represents a valid abstract state $\mathcal{S}_a$. For instance, φ may need $\mathcal{S}$ to contain a linked list and relate it to an abstract mathematical set in $\mathcal{S}_a$ for a set object. Besides, φ should always require the client states in $\mathcal{S}$ and $\mathcal{S}_a$ to be identical.

Next we define a contextual refinement between the concrete object and its specification, which is equivalent to linearizability.

Definition 3 (Basic Contextual Refinement). $\Pi \sqsubseteq_\varphi \Pi_A$ *iff*

$$\forall n, C_1, \ldots, C_n, \mathcal{S}, \mathcal{S}_a.\ (\varphi(\mathcal{S}) = \mathcal{S}_a)$$
$$\Longrightarrow \mathcal{O}[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), \mathcal{S}]\!] \subseteq \mathcal{O}[\![(\textbf{let } \Pi_A \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), \mathcal{S}_a]\!]\,.$$

Remember that $\mathcal{O}[\![W, \mathcal{S}]\!]$ represents the prefix-closed set of observable event traces generated during the executions of $(W, \mathcal{S})$, which is defined in Figure 5.

Following Filipović *et al.* [4], we can prove that linearizability is equivalent to this contextual refinement. We give the proofs in the TR [14].

Theorem 4 (Basic Equivalence). $\Pi \preceq_\varphi \Pi_A \iff \Pi \sqsubseteq_\varphi \Pi_A$.

Theorem 4 allows us to use $\Pi \sqsubseteq_\varphi \Pi_A$ to identify linearizable objects. However, we cannot use it to characterize progress properties of objects. For the following example, $\Pi \sqsubseteq_\varphi \Pi_A$ holds although no concrete method call of `f` could finish (we assume this object contains a method `f` only).

Π(`f`) : `while(true) skip;` $\quad$ Π_A(`f`) : `skip;` $\quad$ C : `print(1); f(); print(1);`

The reason is that $\Pi \sqsubseteq_\varphi \Pi_A$ considers a *prefix-closed* set of event traces at the abstract side. For the above client C, the observable behaviors of **let** Π **in** C can all be found in the prefix-closed set of behaviors produced by **let** Π_A **in** C.

4 Formalizing Progress Properties

We define progress in Figure 6 as properties over both event traces T and object implementations Π. We say an object implementation Π has a progress property P iff all its event traces have the property. Here we use $\mathcal{T}_\omega$ to generate the event traces. Its definition in Figure 6 is similar to $\mathcal{T}[\![W, \mathcal{S}]\!]$ of Figure 5, but $\mathcal{T}_\omega[\![W, \mathcal{S}]\!]$ is for the set of finite or infinite event traces produced by *complete* executions. We use $(W, \mathcal{S}) \xrightarrow{T}{}^{\omega} \cdot$ to denote the existence of a T-labelled infinite execution. $(W, \mathcal{S}) \xrightarrow{T}{}^{*} (\textbf{skip}, \_)$ represents a terminating execution that produces T. By using $\lfloor W \rfloor$, we append **end** at the end of each thread to explicitly mark the

Definition. An object Π satisfies P under a refinement mapping φ, $P_\varphi(\Pi)$, iff $\forall n, C_1, \ldots, C_n, \mathcal{S}, T.\ T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \| \ldots \| C_n), \mathcal{S}]\!] \wedge (\mathcal{S} \in dom(\varphi)) \implies P(T)$.

$$\mathcal{T}_\omega[\![W, \mathcal{S}]\!] \stackrel{\text{def}}{=} \{(\mathbf{spawn}, |W|) :: T \mid (\lfloor W \rfloor, \mathcal{S}) \stackrel{T}{\longmapsto}{}^{\omega} \cdot \vee (\lfloor W \rfloor, \mathcal{S}) \stackrel{T}{\longmapsto}{}^{*} (\mathbf{skip}, \_) \vee (\lfloor W \rfloor, \mathcal{S}) \stackrel{T}{\longmapsto}{}^{*} \mathbf{abort}\}$$

$$\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \| \ldots \| C_n \rfloor \stackrel{\text{def}}{=} \mathbf{let}\ \Pi\ \mathbf{in}\ (C_1; \mathbf{end}) \| \ldots \| (C_n; \mathbf{end})$$

$$|\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \| \ldots \| C_n| \stackrel{\text{def}}{=} n \qquad \mathsf{tnum}((\mathbf{spawn}, n) :: T) \stackrel{\text{def}}{=} n$$

$$\mathsf{pend\_inv}(T) \stackrel{\text{def}}{=} \{e \mid \exists i.\ e = T(i) \wedge \mathsf{is\_inv}(e) \wedge \neg \exists j.\ (j > i \wedge \mathsf{match}(e, T(j)))\}$$

$$\mathsf{prog\text{-}t}(T) \ \text{ iff } \ \forall i, e.\ e \in \mathsf{pend\_inv}(T(1..i)) \implies \exists j.\ j > i \wedge \mathsf{match}(e, T(j))$$

$$\mathsf{prog\text{-}s}(T) \ \text{ iff } \ \forall i, e.\ e \in \mathsf{pend\_inv}(T(1..i)) \implies \exists j.\ j > i \wedge \mathsf{is\_ret}(T(j))$$

$$\mathsf{abt}(T) \ \text{ iff } \ \exists i.\ \mathsf{is\_abt}(T(i))$$

$$\mathsf{sched}(T) \ \text{ iff } \ |T| = \omega \wedge \mathsf{pend\_inv}(T) \neq \emptyset \implies \exists e.\ e \in \mathsf{pend\_inv}(T) \wedge |(T|_{\mathsf{tid}(e)})| = \omega$$

$$\mathsf{fair}(T) \ \text{ iff } \ |T| = \omega \implies \forall \mathsf{t} \in [1..\mathsf{tnum}(T)].\ |(T|_{\mathsf{t}})| = \omega \vee \mathsf{last}(T|_{\mathsf{t}}) = (\mathsf{t}, \mathbf{term})$$

$$\mathsf{iso}(T) \ \text{ iff } \ |T| = \omega \implies \exists \mathsf{t}, i.\ (\forall j.\ j \geq i \implies \mathsf{tid}(T(j)) = \mathsf{t})$$

$$\mathsf{wait\text{-}free} \ \text{ iff } \ \mathsf{sched} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt} \qquad \mathsf{starvation\text{-}free} \ \text{ iff } \ \mathsf{fair} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt}$$

$$\mathsf{lock\text{-}free} \ \text{ iff } \ \mathsf{sched} \implies \mathsf{prog\text{-}s} \vee \mathsf{abt} \qquad \mathsf{deadlock\text{-}free} \ \text{ iff } \ \mathsf{fair} \implies \mathsf{prog\text{-}s} \vee \mathsf{abt}$$

$$\mathsf{obstruction\text{-}free} \ \text{ iff } \ \mathsf{sched} \wedge \mathsf{iso} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt}$$

Fig. 6. Formalizing Progress Properties

$$\mathsf{lock\text{-}free} \iff \mathsf{wait\text{-}free} \vee \mathsf{prog\text{-}s} \qquad \mathsf{starvation\text{-}free} \iff \mathsf{wait\text{-}free} \vee \neg\mathsf{fair}$$

$$\mathsf{obstruction\text{-}free} \iff \mathsf{lock\text{-}free} \vee \neg\mathsf{iso} \qquad \mathsf{deadlock\text{-}free} \iff \mathsf{lock\text{-}free} \vee \neg\mathsf{fair}$$

Fig. 7. Relationships between Progress Properties

termination of the thread. We also insert the spawning event $(\mathbf{spawn}, n)$ at the beginning of T, where n is the number of threads in W. Then we can use $\mathsf{tnum}(T)$ to get the number n, which is needed to define fairness, as shown below.

Before formulating each progress property over event traces, we first define some auxiliary properties in Figure 6. $\mathsf{prog\text{-}t}(T)$ guarantees that every method call in T eventually finishes. $\mathsf{prog\text{-}s}(T)$ guarantees that *some* pending method call finishes. Different from prog-t, the return event $T(j)$ in prog-s does not have to be a matching return of the pending invocation e. $\mathsf{abt}(T)$ says that T ends with a fault event.

There are three useful conditions on scheduling. The basic requirement for a good schedule is sched. If T is infinite and there exist pending calls, then at least one pending thread should be scheduled infinitely often. In fact, there are two possible reasons causing a method call of thread t to pend. Either t is no longer scheduled, or it is always scheduled but the method call never finishes. sched rules out the bad schedule where no thread with an invoked method is active. For instance, the following infinite trace does *not* satisfy sched.

$$\mathsf{div\_tids}(T) \stackrel{\text{def}}{=} \{\mathsf{t} \mid (\,|(T|_{\mathsf{t}})| = \omega\,)\}$$
$$\mathcal{O}_{\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!]\}$$
$$\mathcal{O}_{i\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!] \wedge \mathsf{iso}(T)\}$$
$$\mathcal{O}_{f\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!] \wedge \mathsf{fair}(T)\}$$
$$\mathcal{O}_{t\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{(\mathsf{get\_obsv}(T), \mathsf{div\_tids}(T)) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!]\}$$
$$\mathcal{O}_{ft\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{(\mathsf{get\_obsv}(T), \mathsf{div\_tids}(T)) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!] \wedge \mathsf{fair}(T)\}$$

Fig. 8. Generation of Complete Event Traces

$$(\mathsf{t}_1, f_1, n_1) :: (\mathsf{t}_2, f_2, n_2) :: (\mathsf{t}_1, \mathbf{obj}) :: (\mathsf{t}_3, \mathbf{clt}) :: (\mathsf{t}_3, \mathbf{clt}) :: (\mathsf{t}_3, \mathbf{clt}) :: \ldots$$

If T is infinite, $\mathsf{fair}(T)$ requires every non-terminating thread be scheduled infinitely often; and $\mathsf{iso}(T)$ requires eventually only one thread be scheduled. We can see that a fair schedule is a good schedule satisfying sched.

At the bottom of Figure 6 we define the progress properties formally. We omit the parameter T in the formulae to simplify the presentation. An event trace T is wait-free (*i.e.*, $\mathsf{wait\text{-}free}(T)$ holds) if under the good schedule sched, it guarantees prog-t unless it ends with a fault. $\mathsf{lock\text{-}free}(T)$ is similar except that it guarantees prog-s. Starvation-freedom and deadlock-freedom guarantee prog-t and prog-s under fair scheduling. Obstruction-freedom guarantees prog-t if some pending thread is always scheduled (sched) and runs in isolation (iso).

Figure 7 contains lemmas that relate progress properties. For instance, an event trace is starvation-free, iff it is wait-free or not fair. These lemmas give us the relationship lattice in Figure 1. To close the lattice, we also define a progress property in the sequential setting. *Sequential termination* guarantees that every method call must finish in a trace produced by a sequential client. The formal definition is given in the companion TR [14], and we prove that it is implied by each of the five progress properties for concurrent objects.

5 Equivalence to Contextual Refinements

We extend the basic contextual refinement in Definition 3 to observe progress as well as linearizability. For each progress property, we carefully choose the observable behaviors at the concrete and the abstract levels.

5.1 Observable Behaviors

In Figure 8, we define various observable behaviors for the termination-sensitive contextual refinements.

We use $\mathcal{O}_{\omega}[\![W,\mathcal{S}]\!]$ to represent the set of observable event traces produced by complete executions of $(W,\mathcal{S})$. Recall that $\mathsf{get\_obsv}(T)$ gets the sub-trace of T consisting of all the observable events only. Unlike the prefix-closed set $\mathcal{O}[\![W,\mathcal{S}]\!]$, this definition utilizes $\mathcal{T}_{\omega}[\![W,\mathcal{S}]\!]$ (see Figure 6) whose event traces are all complete and could be infinite. Thus it allows us to observe divergence of the

Table 2. Contextual Refinements $\Pi \sqsubseteq^{P}_{\varphi} \Pi_A$ for Progress Properties P

P	wait-free	lock-free	obstruction-free	deadlock-free	starvation-free
$\Pi \sqsubseteq^{P}_{\varphi} \Pi_A$	$\mathcal{O}_{t\omega} \subseteq \mathcal{O}_{t\omega}$	$\mathcal{O}_{\omega} \subseteq \mathcal{O}_{\omega}$	$\mathcal{O}_{i\omega} \subseteq \mathcal{O}_{\omega}$	$\mathcal{O}_{f\omega} \subseteq \mathcal{O}_{\omega}$	$\mathcal{O}_{ft\omega} \subseteq \mathcal{O}_{t\omega}$

whole program. $\mathcal{O}_{i\omega}$ and $\mathcal{O}_{f\omega}$ take the complete observable traces of *isolating* and fair executions respectively. Here $\mathsf{iso}(T)$ and $\mathsf{fair}(T)$ are defined in Figure 6.

We could also observe divergence of individual threads rather than the whole program. We define $\mathsf{div\_tids}(T)$ to collect the set of threads that diverge in the trace T. Then we write $\mathcal{O}_{t\omega}[\![W, \mathcal{S}]\!]$ to get both the observable behaviors and the diverging threads in the complete executions. $\mathcal{O}_{ft\omega}[\![W, \mathcal{S}]\!]$ is defined similarly but considers fair executions only.

More on divergence. In general, divergence means non-termination. For example, we could say that the following two-threaded program (5.1) must diverge since it never terminates.

```
x := x + 1;    ||    while(true) skip;          (5.1)
```

But for individual threads, divergence is not equivalent to non-termination, since a non-terminating thread may either have an infinite execution or simply be not scheduled from some point due to unfair scheduling. We view only the former case as divergence. For instance, in an unfair execution, the left thread of (5.1) may never be scheduled and hence it has no chance to terminate. It does not diverge. Similarly, for the following program (5.2),

```
while(true) skip;    ||    while(true) skip;     (5.2)
```

the whole program must diverge, but it is possible that a single thread does not diverge in an execution.

5.2 New Contextual Refinements and Equivalence Results

In Table 2, we summarize the definitions of the termination-sensitive contextual refinements. Each new contextual refinement follows the basic one in Definition 3 but takes different observable behaviors as specified in Table 2. For example, the contextual refinement for wait-freedom is formally defined as follows:

$$\Pi \sqsubseteq^{\mathsf{wait\text{-}free}}_{\varphi} \Pi_A \;\; \text{iff} \;\; (\forall n, C_1, \ldots, C_n, \mathcal{S}, \mathcal{S}_a.\ (\varphi(\mathcal{S}) = \mathcal{S}_a) \Longrightarrow \mathcal{O}_{t\omega}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), \mathcal{S})]\!] \subseteq \mathcal{O}_{t\omega}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), \mathcal{S}_a]\!]).$$

Theorem 5 says that linearizability with a progress property P together is equivalent to the corresponding contextual refinement $\sqsubseteq^{P}_{\varphi}$.

Theorem 5 (Equivalence). *$\Pi \preceq_{\varphi} \Pi_A \wedge P_{\varphi}(\Pi) \iff \Pi \sqsubseteq^{P}_{\varphi} \Pi_A$, where P is* wait-free, lock-free, obstruction-free, deadlock-free *or* starvation-free.

Here we assume the object specification Π_A is *total*, *i.e.*, the abstract operations never block. We provide the proofs of our equivalence results in the TR [14].

The contextual refinement for wait-freedom takes $\mathcal{O}_{t\omega}$ at both the concrete and the abstract levels. The divergence of individual threads as well as I/O events are treated as observable behaviors. The intuition of the equivalence is as

follows. Since a wait-free object Π guarantees that every method call finishes, we have to blame the client code itself for the divergence of a thread using Π. That is, even if the thread uses the abstract object Π_A, it must still diverge.

As an example, consider the client program (2.1). Intuitively, for any execution in which the client uses the abstract operations, only the right thread $\mathtt{t}_2$ diverges. Thus $\mathcal{O}_{t\omega}$ of the abstract program is a singleton set $\{(\epsilon, \{\mathtt{t}_2\})\}$. When the client uses the wait-free object in Figure 2(a), its $\mathcal{O}_{t\omega}$ set is still $\{(\epsilon, \{\mathtt{t}_2\})\}$. It does not produce more observable behaviors. But if it uses a non-wait-free object (such as the one in Figure 2(b)), the left thread $\mathtt{t}_1$ does not necessarily finish. The $\mathcal{O}_{t\omega}$ set becomes $\{(\epsilon, \{\mathtt{t}_2\}), (\epsilon, \{\mathtt{t}_1, \mathtt{t}_2\})\}$. It produces more observable behaviors than the abstract client, breaking the contextual refinement. Thanks to observing div_tids that collects the diverging threads, we can rule out non-wait-free objects which may cause more threads to diverge.

$\Pi \sqsubseteq_{\varphi}^{\text{lock-free}} \Pi_A$ takes coarser observable behaviors. We observe the divergence of the whole client program by using $\mathcal{O}_\omega$ at both the concrete and the abstract levels. Intuitively, a lock-free object Π ensures that some method call will finish, thus the client using Π diverges only if there are an infinite number of method calls. Then it must also diverge when using the abstract object Π_A.

For example, consider the client (2.1). The whole client program diverges in every execution both when it uses the lock-free object in Figure 2(b) and when it uses the abstract one. The $\mathcal{O}_\omega$ set of observable behaviors is $\{\epsilon\}$ at both levels. On the other hand, the following client must terminate and print out both 1 and 2 in every execution. The $\mathcal{O}_\omega$ set is $\{1::2::\epsilon, 2::1::\epsilon\}$ at both levels.

```
inc(); print(1);   ||   dec(); print(2);        (5.3)
```

Instead, if the client (5.3) uses the non-lock-free object in Figure 2(c), it may diverge and nothing is printed out. The $\mathcal{O}_\omega$ set becomes $\{\epsilon, 1::2::\epsilon, 2::1::\epsilon\}$, which contains more behaviors than the abstract side. Thus $\Pi \sqsubseteq_{\varphi}^{\text{lock-free}} \Pi_A$ fails.

Obstruction-freedom ensures progress for isolating executions in which eventually only one thread is running. Correspondingly, $\Pi \sqsubseteq_{\varphi}^{\text{obstruction-free}} \Pi_A$ restricts our considerations to isolating executions. It takes $\mathcal{O}_{i\omega}$ at the concrete level and $\mathcal{O}_\omega$ at the abstract level.

To understand the equivalence, consider the client (5.3) again. For isolating executions with the obstruction-free object in Figure 2(c), it *must* terminate and print out both 1 and 2. The $\mathcal{O}_{i\omega}$ set at the concrete level is $\{1::2::\epsilon, 2::1::\epsilon\}$, the same as the set $\mathcal{O}_\omega$ of the abstract side. Non-obstruction-free objects in general do not guarantee progress for some isolating executions. If the client uses the object in Figure 2(d) or (e), the $\mathcal{O}_{i\omega}$ set is $\{\epsilon, 1::2::\epsilon, 2::1::\epsilon\}$, not a subset of the abstract $\mathcal{O}_\omega$ set. The undesired empty observable trace is produced by unfair executions, where a thread acquires the lock and gets suspended and then the other thread would keep requesting the lock forever (it is executed in isolation).

$\Pi \sqsubseteq_{\varphi}^{\text{deadlock-free}} \Pi_A$ uses $\mathcal{O}_{f\omega}$ at the concrete side, ruling out undesired divergence caused by unfair scheduling. For the client (5.3) with the object in Figure 2(d) or (e), its $\mathcal{O}_{f\omega}$ set is same as the set $\mathcal{O}_\omega$ at the abstract level.

For $\Pi \sqsubseteq_{\varphi}^{\text{starvation-free}} \Pi_A$, we still consider only fair executions at the concrete level (similar to deadlock-freedom), but observe the divergence of individual

threads rather than the whole program (similar to wait-freedom). It uses $\mathcal{O}_{ft\omega}$ at the concrete side and $\mathcal{O}_{t\omega}$ at the abstract level. For the client (5.3) with the starvation-free object in Figure 2(e), no thread diverges in any fair execution. Then the set $\mathcal{O}_{ft\omega}$ of observable behaviors is $\{(1::2::\epsilon, \emptyset), (2::1::\epsilon, \emptyset)\}$, which is same as the set $\mathcal{O}_{t\omega}$ at the abstract level.

Observing threaded divergence allows us to distinguish starvation-free objects from deadlock-free objects. Consider the client (2.1). Under fair scheduling, we know only the right thread $\mathtt{t}_2$ would diverge when using the starvation-free object in Figure 2(e). The set $\mathcal{O}_{ft\omega}$ is $\{(\epsilon, \{\mathtt{t}_2\})\}$. It coincides with the abstract behaviors $\mathcal{O}_{t\omega}$. But when using the deadlock-free object of Figure 2(d), the $\mathcal{O}_{ft\omega}$ set becomes $\{(\epsilon, \{\mathtt{t}_2\}), (\epsilon, \{\mathtt{t}_1, \mathtt{t}_2\})\}$, breaking the contextual refinement.

6 Related Work and Conclusion

There is a large body of work discussing the five progress properties and the contextual refinements individually. Our work in contrast studies their relationships, which have not been considered much before.

Gotsman and Yang [6] propose a new linearizability definition that preserves lock-freedom, and suggest a connection between lock-freedom and a termination-sensitive contextual refinement. We do not redefine linearizability here. Instead, we propose a unified framework to systematically relate all the five progress properties plus linearizability to various contextual refinements.

Herlihy and Shavit [10] informally discuss all the five progress properties. Our definitions in Section 4 mostly follow their explanations, but they are more formal and close the gap between program semantics and their history-based interpretations. We also notice that their obstruction-freedom is inappropriate for some examples (see TR [14]), and propose a different definition that is closer to the common intuition [9]. In addition, we relate the progress properties to contextual refinements, which consider the extensional effects on client behaviors.

Fossati *et al.* [5] propose a uniform approach in the π-calculus to formulate both the standard progress properties and their observational approximations. Their technical setting is completely different from ours. Also, their observational approximations for lock-freedom and wait-freedom are strictly weaker than the standard notions. Their deadlock-freedom and starvation-freedom are not formulated, and there is no observational approximation given for obstruction-freedom. In comparison, our framework relates each of the five progress properties (plus linearizablity) to an *equivalent* contextual refinement.

There are also formulations of progress properties based on temporal logics. For example, Petrank *et al.* [15] formalize the three non-blocking properties and Dongol [3] formalize all the five progress properties, using linear temporal logics. Those formulations make it easier to do model checking (*e.g.*, Petrank *et al.* [15] also build a tool to model check a variant of lock-freedom), while our contextual refinement framework is potentially helpful for modular Hoare-style verification.

Conclusion. We have introduced a contextual refinement framework to unify various progress properties. For linearizable objects, each progress property is

equivalent to a specific termination-sensitive contextual refinement, as summarized in Table 1. The framework allows us to verify safety and liveness properties of client programs at a high abstraction level by replacing concrete method implementations with abstract operations. It also makes it possible to borrow ideas from existing proof methods for contextual refinements to verify linearizability and a progress property together, which we leave as future work.

Acknowledgments. We would like to thank anonymous referees for their helpful suggestions and comments. This work is supported in part by China Scholarship Council, NSFC grants 61073040 and 61229201, NCET grant NCET-2010-0984, and the Fundamental Research Funds for the Central Universities (Grant No. WK0110000018). It is also supported in part by DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, ONR grant N000141210478, and NSF grants 0915888 and 1065451. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

1. Aspnes, J., Herlihy, M.: Wait-free data structures in the asynchronous PRAM model. In: SPAA, pp. 340–349 (1990)
2. Birkedal, L., Sieczkowski, F., Thamsborg, J.: A concurrent logical relation. In: CSL, pp. 107–121 (2012)
3. Dongol, B.: Formalising progress properties of non-blocking programs. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 284–303. Springer, Heidelberg (2006)
4. Filipovic, I., O'Hearn, P., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theor. Comput. Sci. 411(51-52), 4379–4398 (2010)
5. Fossati, L., Honda, K., Yoshida, N.: Intensional and extensional characterisation of global progress in the π-calculus. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 287–301. Springer, Heidelberg (2012)
6. Gotsman, A., Yang, H.: Liveness-preserving atomicity abstraction. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 453–465. Springer, Heidelberg (2011)
7. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1), 124–149 (1991)
8. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: ICDCS, pp. 522–529 (2003)
9. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (April 2008)
10. Herlihy, M., Shavit, N.: On the nature of progress. In: Fernàndez Anta, A., Lipari, G., Roy, M. (eds.) OPODIS 2011. LNCS, vol. 7109, pp. 313–328. Springer, Heidelberg (2011)
11. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
12. Liang, H., Feng, X.: Modular verification of linearizability with non-fixed linearization points. In: PLDI (to appear, 2013)
13. Liang, H., Feng, X., Fu, M.: A rely-guarantee-based simulation for verifying concurrent program transformations. In: POPL, pp. 455–468 (2012)
14. Liang, H., Hoffmann, J., Feng, X., Shao, Z.: The extended version of the present paper (2013), `http://kyhcs.ustcsz.edu.cn/relconcur/prog`
15. Petrank, E., Musuvathi, M., Steensgaard, B.: Progress guarantee for parallel programs via bounded lock-freedom. In: PLDI, pp. 144–154 (2009)

Aspect-Oriented Linearizability Proofs

Thomas A. Henzinger[1], Ali Sezgin[1], and Viktor Vafeiadis[2]

[1] IST Austria
{tah,asezgin}@ist.ac.at
[2] MPI-SWS
viktor@mpi-sws.org

Abstract. Linearizability of concurrent data structures is usually proved by monolithic simulation arguments relying on identifying the so-called linearization points. Regrettably, such proofs, whether manual or automatic, are often complicated and scale poorly to advanced non-blocking concurrency patterns, such as helping and optimistic updates.

In response, we propose a more modular way of checking linearizability of concurrent queue algorithms that does not involve identifying linearization points. We reduce the task of proving linearizability with respect to the queue specification to establishing four basic properties, each of which can be proved independently by simpler arguments. As a demonstration of our approach, we verify the Herlihy and Wing queue, an algorithm that is challenging to verify by a simulation proof.

1 Introduction

Linearizability [8] is widely accepted as the standard correctness requirement for concurrent data structure implementations. It amounts to showing that all methods are atomic and obey the high-level sequential specification of the data structure. For example, an unbounded queue must support the following two methods: *enqueue*, which extends the queue by appending one element to its end, and *dequeue*, which removes and returns the first element of the queue.

The standard way to prove that a concurrent queue implementation is linearizable is to prove an invariant which relates the state of the implementation to the state of the specification. A well-established approach (e.g. [1–5, 11, 13–15]) is to identify the linearization points, which when performed by the implementation change the state of the specification, and to then construct a forward or backward simulation.

While for a number of concurrent algorithms, spotting the linearization points may be straightforward (and has even been automated to some extent [15]), in general specifying the linearization points can be very difficult. For instance, in implementations using a helping mechanism, they can lie in code not syntactically belonging to the thread and operation in question, and can even depend on future behavior. There are numerous examples in the literature, where this is the case; to mention only a few concurrent queues: the Herlihy and Wing queue [8], the optimistic queue [10], the elimination queue [12], the baskets queue [9], the flat-combining queue [6].

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 242–256, 2013.

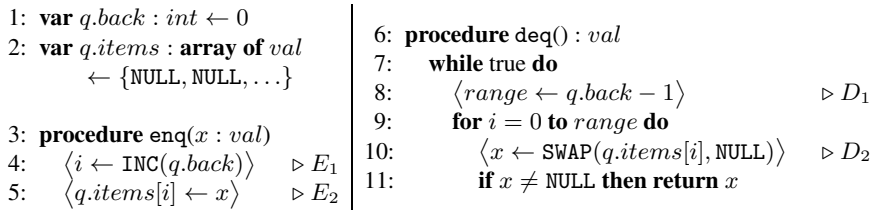


Fig. 1. Herlihy and Wing queue [8]

The HW Queue. In this paper, we focus on the Herlihy and Wing queue [8] (henceforth, HW queue for short) that illustrates nicely the difficulties encountered when defining a simulation relation based on linearization points. The code is given in Fig. 1. The queue is represented as a pre-allocated unbounded array, $q.items$, initially filled with NULLs, and a marker, $q.back$, pointing to the end of the used part of the array. Enqueuing an element is done in two steps: the marker to the end of the array is incremented (E_1), thereby reserving a slot for storing the element, and then the element is stored at the reserved slot (E_2). Dequeue is more complex: it reads the marker (D_1), and then searches from the beginning of the array up to the marker to see if it contains a non-NULL element. It removes and returns the first such element it finds (D_2). If no element is found, dequeue starts again afresh. Each of the four statements surrounded by $\langle\rangle$ brackets and annotated by E_i or D_i for $i = 1, 2$ is assumed to execute in isolation.

Consider the following execution fragment, where $\cdot$ denotes context switches between concurrent threads,

$$(t : E_1) \cdot (u : E_1) \cdot (v : D_1, D_2) \cdot (u : E_2) \cdot (t : E_2) \cdot (w : D_1)$$

which have threads t and u executing enqueue instances, v and w executing dequeue instances. At the end of this fragment, v is ready to dequeue the element enqueued by u, and w is ready to dequeue the element enqueued by t. In order to define a simulation relation from this interleaving sequence to a valid sequential queue behavior, where operations happen in isolation, we have to choose the linearization points for the two completed enqueue instances. The difficulty lies in the fact that no matter which statements are chosen as the linearization points for the two enqueue instances, there is always an extension to the fragment inconsistent with the particular choice of linearization points. For instance, if we choose $(t : E_1)$ as the linearization point for t, then the extension

$$(v : D_2, \textbf{return}) \cdot (z : D_1, D_2, \textbf{return})$$

requiring u's element be enqueued before that of t's, will be inconsistent. If on the other hand, any statement which makes u linearize before t, then the extension

$$(w : D_2, \textbf{return}) \cdot (z : D_1, D_2, D_2, \textbf{return})$$

requiring the reverse order of enqueueing will be inconsistent. This shows not only that finding the correct linearization sequence can be challenging, but also that the simulation proofs will require to reason about the entire state of the system, as the local state of one thread can affect the linearization of another.

Our Contribution. In our experience, this and similar tricks for reducing synchronization among threads so as to achieve better performance, make concurrent algorithms extremely difficult to reason about when one is constrained to establishing a simulation relation. However, if two methods overlap in time, then the only thing enforced by linearizability is that their effects are observed in *some* and same order by all threads. For instance, in the example given above, the simple answer for the particular ordering between the linearization points of the enqueue instances of t and u, is that it does not matter! As long as enqueue instances overlap, their values can be dequeued in any order.

Building on this observation, our contribution is to simplify linearizability proofs by modularizing them. We reduce the task of proving linearizability to establishing four relatively simple properties, each of which may be reasoned about independently. In (loose) analogy to aspect-oriented programming, we are proposing "aspect-oriented" linearizability proofs for concurrent queues, where each of these four properties will be proved independently.

So what are these properties? A correct (i.e., linearizable) concurrent queue:

(1) must not allow dequeuing an element that was never enqueued;
(2) it must not allow the same element to be dequeued twice;
(3) it must not allow elements to be dequeued out of order; and
(4) it must correctly report whether the queue is empty or not.

Although similar properties were already mentioned by Herlihy and Wing [8], we for the first time prove that suitably formalized versions of these four properties are not only necessary, but also sufficient, conditions for linearizability with respect to the queue specification, at least for what we call *purely-blocking* implementations. This is a rather weak requirement satisfied by all non-blocking methods, as well as by possibly blocking methods, such as HW `deq()` method, whose blocking executions do not modify the global state.

The rest of the paper is structured as follows: Section 2 recalls the definition of linearizability in terms of execution histories; Section 3 formalizes the aforementioned four properties, and proves that they are necessary and sufficient conditions for proving linearizability of queues; Section 4 returns to the HW queue example and presents a detailed manual proof of its correctness; and Section 5 explains how the bulk of this proof was also performed automatically by an adaptation of CAVE [15]. Finally, in Sec. 6 we discuss related work, and in §7 we conclude.

2 Technical Background

In this section, we introduce common notations that will be used throughout the paper and recall the definition of linearizability.

Histories, Linearizability. For any function f from A to B and $A' \subseteq A$, let $f(A') \stackrel{\text{def}}{=} \{f(a) \mid a \in A'\}$. Given two sequences x and y, let $x \cdot y$ denote their concatenation, and let $x \sim_{\text{perm}} y$ hold if one is a permutation of the other.

A *data structure* $\mathcal{D}$ is a pair $(D, \Sigma_{\mathcal{D}})$, where D is the *data domain* and $\Sigma_{\mathcal{D}}$ is the *method alphabet*. An *event* of $\mathcal{D}$ is a triple (m, d_i, d_o), for some $m \in \Sigma_{\mathcal{D}}$, $d_1, d_2 \in D$.

Intuitively, (m, d_i, d_o) denotes the application of method m with input argument d_i returning the output value d_o. A sequence over events of $\mathcal{D}$ is called a *behavior*. The *semantics* of data structure $\mathcal{D}$ is a set of behaviors, called *legal* behaviors.

Each event $a = (m, d_i, d_o)$ generates two *actions*: the *invocation* of a, written as $inv(a)$, and the *response* of a, written as $res(a)$. We will also use $m_i(d_i)$ and $m_r(d_o)$ to denote the invocation and the response actions, respectively. When a particular method m does not have an input (resp., output) parameter, we will write $(m, \perp, x)$ (resp., $(m, x, \perp)$), and $m_i()$ (resp., $m_r()$) for the corresponding invocation (resp., response) action.

In this paper, a *history* of $\mathcal{D}$ is a sequence of invocation and response actions of $\mathcal{D}$. We will assume the existence of an implicit identifier in each history c that uniquely pairs each invocation with its corresponding response action, if the latter also occurs in c. A history c is *well-formed* if every response action occurs after its associated invocation action in c. We will consider only well-formed histories. An event is *completed* in c, if both of its invocation and response actions occur in c. An event is *pending* in c, if only its invocation occurs in c. We define $remPending(c)$ to be the sub-sequence of c where all pending events have been removed. An event e precedes another event e' in c, written $e \prec_c e'$, if the response of e occurs before the invocation of e' in c. For event e, $Before(e, c)$ denotes the set of all events that precede e in c. Similarly, $After(e, c)$ denotes the set of all events that are preceded by e in c. Formally,

$$Before(e, c) \stackrel{\text{def}}{=} \{e' \mid e' \prec_c e\} \qquad \text{and} \qquad After(e, c) \stackrel{\text{def}}{=} \{e' \mid e \prec_c e'\}.$$

History c is called *complete* if it does not have any pending events. For a possibly incomplete history c, a *completion* of c, written $\hat{c}$, is a (well-formed) complete history such that $\hat{c} = remPending(c \cdot c')$ where c' contains only response events. Let $Compl(c)$ denote the set of all completions of c.

A history is called *sequential* if all invocations in c are immediately followed by their matching responses, with the possible exception of the very last action which can only be the invocation of a pending event. We identify complete sequential histories with behaviors of $\mathcal{D}$ by mapping each consecutive pair of matching actions in the former to its event constructing the latter. A sequential history s is a *linearization* of a history c, if there exists $\hat{c} \in Compl(c)$ such that $\hat{c} \sim_{\text{perm}} s$ and whenever $e \prec_{\hat{c}} e'$ we have $e \prec_s e'$.

Definition 1 (Linearizability [8]). *A set of histories C is linearizable with respect to a data structure $\mathcal{D}$, if for any $c \in C$, there exists a linearization of c which is a legal behavior of $\mathcal{D}$.*

Queues. The method alphabet Σ_Q of a queue is the set $\{\texttt{enq}, \texttt{deq}\}$. We will take the data domain to be the set of natural numbers, $\mathbb{N}$, and a distinguished symbol `NULL` not in $\mathbb{N}$. Events are written as $\texttt{enq}(x)$, short for $(\texttt{enq}, x, \perp)$, and $\texttt{deq}(x)$, short for $(\texttt{deq}, \perp, x)$. Events with `enq` are called *enqueue* events, and those with `deq` are called *dequeue* events.

Let c be a history. $Enq(c)$ denotes the set of all enqueue events invoked (and not necessarily completed) in c. Similarly, $Deq(c)$ denotes the set of all dequeue events invoked in c. A set $A \subseteq Enq(c) \cup Deq(c)$ is *closed under* $\prec_c$ if $a \in A$ and $b \prec_c a$, then $b \in A$.

For an enq event e in c, $Val_c(e)$ denotes the value to be inserted by e in c. Formally, $Val_c(\texttt{enq}(x)) = x$. Similarly, for a completed deq event d in c, $Val_c(d)$ denotes the value removed by d in c. Formally, $Val_c(\texttt{deq}(x)) = x$. For a pending deq event, $Val_c(\texttt{deq}(x))$ is undefined.

We will use a labelled transition system, LTS_Q, to define the queue semantics. The states of LTS_Q are sequences over $\mathbb{N}$, the initial state is the empty sequence ε. There is a transition from q to q' with action a, written $q \xrightarrow{a} q'$, if (*i*) $a = \texttt{enq}(x)$ and $q' = q \cdot x$, or (*ii*) $a = \texttt{deq}(x)$ and $q = x' \cdot q'$, or (*iii*) $a = \texttt{deq}(\texttt{NULL})$ and $q = q' = \varepsilon$. A queue is *partial* if the last transition (NULL returning dequeue event) is not allowed.

A *run* of LTS_Q is an alternating sequence $q_0 l_1 q_1 \ldots l_n q_n$ of states and queue events such that for all $1 \leq i \leq n$, we have $q_{i-1} \xrightarrow{l_i} q_i$. The trace of a run is the sequence $l_1 \ldots l_n$ of the events occurring on the run. A queue behavior b is *legal* iff there is a run of LTS_Q with trace b.

We find it useful to express the queue semantics in an alternative formulation.

Definition 2. *A queue behavior b has a* sequential witness *if there is a total mapping μ_{seq} from $Deq(b)$ to $Enq(b) \cup \{\bot\}$ such that*

- $\mu_{\text{seq}}(d) = e$ *implies* $Val_b(d) = Val_b(e)$,
- $\mu_{\text{seq}}(d) = \bot$ *iff* $Val_b(d) = \texttt{NULL}$,
- $\mu_{\text{seq}}(d) = \mu_{\text{seq}}(d') \neq \bot$ *implies* $d = d'$,
- $e \prec_b e'$ *and there exists* d' *with* $\mu_{\text{seq}}(d') = e'$ *imply* $\mu_{\text{seq}}^{-1}(e) \prec_b d'$,
- $\mu_{\text{seq}}(d) = \bot$ *implies that*
  $|\{e \in Enq(b) \mid e \prec_b d\}| = |\{d' \in Deq(b) \mid d' \prec_b d \wedge \mu_{\text{seq}}(d') \neq \bot\}|$.

Proposition 1. *A queue behavior b is legal iff b has a sequential witness.*

Proof (Sketch). If b is legal, then, by definition, it has a run r in LTS_Q with trace b. Let d be a dequeue event occurring in b. Then there is a transition $q \xrightarrow{d} q'$ in r. If $d = \texttt{deq}(x)$ for some $x \in \mathbb{N}$, then set $\mu_{\text{seq}}(d) = e$ where e is the enqueue event $\texttt{enq}(x)$ which has inserted x into the state sequence. If $d = \texttt{deq}(\texttt{NULL})$, then set $\mu_{\text{seq}}(d) = \bot$. Then, it is easy to check that μ_{seq} satisfies all the conditions of being a sequential witness for b.

For the other direction, let μ_{seq} be a sequential witness for b. We observe that i) an element x is in state q iff an enqueue event $\texttt{enq}(x)$ has happened on the prefix of the run ending at q and the dequeue event with $\mu_{\text{seq}}(d) = e$ has not happened on the same prefix, ii) for any two enqueue events e, e' with $e \prec_b e'$, $Val_b(e)$ occurs in a state before $Val_b(e')$, iii) the relative ordering of inserted elements in a state does not change as long as both are in the state, iv) each enqueue event inserts exactly one element to the state, v) each dequeue event $\texttt{deq}(x)$ with $x \neq \texttt{NULL}$ removes exactly one element from the state, and vi) the dequeue event $\texttt{deq}(\texttt{NULL})$ does not change the state. Then, by induction on the length of b, we show that b has a run in LTS_Q. □

3 Conditions for Queue Linearizability

3.1 Generic Necessary and Sufficient Conditions

We start by reducing the problem of checking linearizability of a given history, c, with respect to the queue specification to finding a mapping from its dequeue events to its

enqueue events satisfying certain conditions. Intuitively, we map each dequeue event to the enqueue event whose value the dequeue removed, or to nothing if the dequeue event returns NULL. We say that the mapping is *safe* if it pairs each deq event with a proper enq event, implying that elements are inserted exactly once and removed at most once. A safe mapping is *ordered* if it additionally respects precedence induced by c. Finally, an ordered mapping is a *linearizability witness* if all NULL returning deq events see at least one state where the queue is logically empty. Below, we formalize these notions.

Definition 3 (Safe Mapping). *A mapping $Match$ from $Deq(c)$ to $Enq(c) \cup \{\bot\}$ is* safe *for c if*
(1) for all $d \in Deq(c)$, if $Match(d) \neq \bot$, then $Val_c(d) = Val_c(Match(d))$;
(2) for all $d \in Deq(c)$, $Match(d) = \bot$ iff $Val_c(d) =$ NULL; and
(3) for all $d, d' \in Deq(c)$, if $Match(d) = Match(d') \neq \bot$, then $d = d'$.

Definition 4 (Ordered Mapping). *A safe mapping $Match$ for c is* ordered *if*
(1) for all $d \in Deq(c)$, we have $d \not\prec_c Match(d)$; and
(2) for all $d \in Deq(c)$ and $e' \in Enq(c)$, if $e' \prec_c Match(d)$, then there exists $d' \in Deq(c)$ such that $e' = Match(d')$.

Definition 5 (Linearization Witness). *An ordered mapping $Match$ for c is a* linearization witness *if for any $d \in Deq(c)$ with $Val_c(d) =$ NULL, there exists a subset $D' \subseteq Deq(c)$ such that $Match(D')$ is closed under $\prec_c$ and $D' \cap After(d, c) = \emptyset$ and $Before(d, c) \cap Enq(c) \subseteq Match(D')$.*

The main result of this section is stated below.

Theorem 1. *A set of histories C is linearizable with respect to queue iff every $c \in C$ has a completion $\hat{c} \in Compl(c)$ that has a linearization witness.*

Proof. **(⇒)** If $c \in C$ is linearizable with respect to queue, then there is a linearization s of c which is a legal queue behavior. By Prop. 1, s has a sequential witness μ_{seq}. The mapping μ_{seq} satisfies the conditions of a linearization witness since all $\prec_c$ orderings are preserved in s.

(⇐) Pick a $c \in C$ and let $\hat{c} \in Compl(c)$ be its completion that has a linearization witness $Match$. Let $<$ be some arbitrary total order on the events of $\hat{c}$. We construct the linearization of $\hat{c}$ inductively as follows:

Let c' be the prefix of $\hat{c}$ that has been processed, and let s' be the resulting sequential history. All events in s' are *placed*. Events that are not placed but are pending after c' are called *candidate*. We extend c' until the first response action that happens after c' in $\hat{c}$. Formally, let $c' \cdot c_e \cdot a_r$ be a prefix of $\hat{c}$ such that c_e contains only invocation actions and a_r is a response action. Let A denote the set of all candidate events after $c' \cdot c_e \cdot a_r$. The new s' is obtained by appending some $a \in A$ as the next event if
(1) a is an enqueue event, and there does not exist another enqueue event e such that $Match^{-1}(e) \prec_{\hat{c}} Match^{-1}(a)$ and e is not placed in s'; or
(2) a is a dequeue event with $Val_{\hat{c}}(a) \neq$ NULL, $Match(a)$ is placed in s', and there does not exist another dequeue event d such that $Match(d) \prec_{\hat{c}} Match(a)$ and d is not placed in s'; or

(3) a is a dequeue event with $Val_{\hat{c}}(a) = \texttt{NULL}$ and the number of enqueue events in s' is equal to the number of dequeue events d with $Val_{\hat{c}}(d) \neq \texttt{NULL}$ in s'.
In case both first and second conditions are satisfied, the candidate element minimal with respect to $<$ is appended to s'. This iteration is repeated until there are no candidate events that satisfy any of the conditions, at which point the inductive step ends with setting c' to $c' \cdot c_e \cdot a_r$. The existence of $Match$ guarantees that such a sequence can be constructed. The constructed sequence s has $Match$ also as a sequential witness, completing the proof. □

3.2 Necessary and Sufficient Conditions for Complete Histories

We now focus on complete histories, namely ones with no pending events. We observe that their linearizability violations can always be manifested in terms of the dequeued values. Intuitively, the possible violations are:

(VFresh)**.** A dequeue event returns a value not inserted by any enqueue event.
(VRepet)**.** Two dequeue events return the value inserted by the same enqueue event.
(VOrd)**.** Two values are enqueued in a certain order, and either they are dequeued in the reverse order or only the later value is dequeued.
(VWit)**.** A dequeue event returning NULL even though the queue is never logically empty during the execution of the dequeue event.

We have the following result which ties the above violation types to linearizable queues.

Proposition 2. *A complete history c has a linearization which is a legal queue behavior iff it has none of the* VFresh*,* VRepet*,* VOrd*,* VWit *violations.*

Proof (Sketch). First, note that as c has no pending events, $Compl(c) = \{c\}$. If c has a linearization which is a legal queue behavior, then by Theorem 1, c has a linearization witness $Match$, and so none of the violations can happen. As $Match$ is safe, (VFresh) and (VRepet) cannot happen; as it is ordered, (VOrd) cannot occur; and as it is a linearization witness, likewise (VWit) cannot happen. Similarly, in the other direction, the absence of all the violations ensures the existence of a linearizability witness. □

We remark that none of the violations mentions the possibility of an element inserted by an enqueue being lost forever. This is intentional, as such histories are ruled out by the following proposition.

Proposition 3. *Given an infinite sequence of complete histories $c_1, c_2, \ldots$ not containing any of the violations above, where for every i, c_i is a prefix of c_{i+1}, and the number of dequeue events in c_i is less than that of c_{i+1}, if c_1 contains an enqueue event* $\texttt{enq}(x)$*, then exists some c_j containing* $\texttt{deq}(x)$.

Proof. We prove this by contradiction. If there is no $\texttt{deq}(x)$ event, then $\texttt{enq}(x)$ is always in the queue, and so, from the absence of VWit violations, none of the dequeue events following $\texttt{enq}(x)$ can return NULL. Also, since dequeue events cannot return values that were not previously enqueued (VFresh) and cannot return the same value multiple times (VRepet), and since the number of dequeue events is increasing, then

there must also be new enqueue events. However, only finitely many of those are not preceded by $\texttt{enq}(x)$ which completes in c_1. This means that eventually one dequeue event has to return an element inserted by $\texttt{enq}(y)$ such that $\texttt{enq}(x) \prec_{c_j} \texttt{enq}(y)$, which is VOrd. □

For checking purposes, we find it useful to re-state the third violation as the following equivalent proof obligation.

(POrd). For any enqueue events e_1 and e_2 with $e_1 \prec_c e_2$ and $Val_c(e_1) \neq Val_c(e_2)$, a dequeue event d_2 cannot return $Val_c(e_2)$ if $Val_c(e_1)$ is not removed in c or is removed by d_1 with $d_2 \prec_c d_1$.

Thus, we need an invariant which specifies all those executions satisfying the premise of POrd, and prove that such an execution cannot end with a dequeue event (in the sense that no other method is preceded by that dequeue event) returning the value of e_2.

3.3 Necessary and Sufficient Conditions for Purely-Blocking Queues

There is a subtle complication in the statement of Theorem 1. The witness mapping is chosen relative to some completion of the concurrent history under consideration. However, because implementations may become blocked, such completions may actually never be reached. This means that one cannot reason about the correctness of a queue implementation by considering only reachable states. What we would ideally like to do is to claim that if the implementation violates linearizability, then there is a finite complete history of the implementation which has no witness. In other words, if the implementation contains an incomplete history with no witness, then that execution is the prefix of a complete history of the implementation.

Let C be the set of all possible execution histories of a library implementation. We call a library implementation *completable* iff for every history $c \in C$, we have $Compl(c) \cap C \neq \emptyset$. For completable implementations, it suffices to consider only complete executions.

Theorem 2. *A completable queue implementation is linearizable iff all its complete histories have none of the* VFresh, VRepet, VOrd *and* VWit *violations.*

Proof. (⇒) If some complete history has a violation, by Prop. 2, it has no linearization, contradicting the assumption that the implementation is linearizable.

(⇐) Consider an arbitrary history c of the implementation. As the implementation is completable, there exists a completion $\hat{c} \in Compl(c)$ that is a valid history of the implementation. From our assumptions, $\hat{c}$ cannot have a violation, and so by Prop. 2, $\hat{c}$ has a linearization, and therefore so does c. □

Since it may not be obvious how to easily prove that an implementation is completable, we introduce the stronger notion of purely-blocking implementations, that is straightforward to check. We say that an implementation is *purely-blocking* when at any reachable state, any pending method, if run in isolation will terminate or its entire execution does not modify the global state.

Proposition 4. *Every purely-blocking implementation is completable.*

Proof. Given a history $c \in C$, we will construct $\hat{c} \in \mathit{Compl}(c) \cap C$. We fix a total order of pending events, and consider them in that order. For a pending method e, if running it in isolation terminates, then extend c with the corresponding response for e. Otherwise, the execution of e does not modify any global state and so can be removed from the history without affecting its realizability. □

We remark that our new notion of purely-blocking is a strictly weaker requirement than the standard non-blocking notions: *obstruction-freedom*, which requires all pending methods to terminate when run in isolation, as well as the stronger notions of lock-freedom and wait-freedom. (See [7] for an in depth exposition of these three notions.)

4 Manually Verifying the Herlihy-Wing Queue

Let us return to the HW queue presented in §1 and prove its correctness manually following our aspect-oriented approach.

First, observe that HW queue is purely-blocking: `enq()` always terminates, and `deq()` can update the global state only by reading $x \neq$ `NULL` at E_2, in which case it immediately terminates. So from Prop. 4 and Theorem 2, it suffices to show that it does not have any of the four violations. The last one, VWit, is trivial as the HW `deq()` never returns `NULL`. So, we are left with three violations whose absence we have to verify: VFresh, VRepet, and VOrd.

Intuitively, there are no VFresh violations because `deq()` can return only a value that has been stored inside the $q.items$ array. The only assignments to $q.items$ are E_1 and D_2: the former can only happen by an $\texttt{enq}(x)$, which puts x into the array; the latter assigns `NULL`.

Likewise, there are no VRepet violations because whenever in an arbitrary history two calls to `deq()` return the same x, then at least twice there was an element of the $q.items$ array holding the value x and was updated to `NULL` by the `SWAP` instruction at D_2. Therefore, at least two assignments of the form $q.items[\_] \leftarrow x$ happened; i.e. there were at least two $\texttt{enq}(x)$ events in the history.

We move on to the more challenging third condition, VOrd. We actually consider its equivalent reformulation, POrd. Fix a value v_2 and consider a history c where every method call enqueuing v_2 is preceded by some method call enqueuing some different value v_1 and there are no `deq()` calls returning v_1 (there may be arbitrarily many concurrent `enq()` and `deq()` calls enqueuing or dequeuing other values). The goal is to show that in this history, no `deq()` return v_2.

Let us suppose there is a dequeue d returning v_2, and try to derive a contradiction. For d to return v_2, it must have read $\mathit{range} \geq i_2$ such that $q.items[i_2] = v_2$. So, d must have read $q.back$ at D_1 after $\texttt{enq}(v_2)$ incremented it at E_1.

Since, $\texttt{enq}(v_1) \prec_c \texttt{enq}(v_2)$, it follows that $\texttt{enq}(v_2)$ will have read a larger value of $q.back$ at E_1 than $\texttt{enq}(v_1)$. So, in particular, once $\texttt{enq}(v_1)$ finishes, the following assertion will hold:

$$\exists i_1 < q.back.\ q.items[i_1] = v_1 \wedge (\forall j < i_1.\ q.items[j] \neq v_2) \qquad (*)$$

$$
\begin{array}{l}
\textbf{procedure } \texttt{deq}(v : val) \\
\quad \textbf{while } \text{true } \textbf{do} \\
\quad\quad \langle range \leftarrow q.back - 1 \rangle \\
\quad\quad \textbf{for } i = 0 \textbf{ to } range \textbf{ do} \\
\quad\quad\quad \left(\begin{array}{l} \left\langle \begin{array}{l} x \leftarrow q.items[i]; \\ \mathsf{assume}(x = v \wedge x \neq \texttt{NULL}); \\ q.items[i] \leftarrow \texttt{NULL} \end{array} \right\rangle ; \\ \textbf{return } x \end{array} \right) \sqcup \left\langle \begin{array}{l} x \leftarrow q.items[i]; \\ \mathsf{assume}(x = \texttt{NULL}); \\ q.items[i] \leftarrow \texttt{NULL} \end{array} \right\rangle
\end{array}
$$

Fig. 2. The HW dequeue method instrumented with the prophecy variable v guessing its return value, where $\sqcup$ stands for non-deterministic choice

Note that since, by assumption, v_1 can never be dequeued, and any later $\texttt{enq}(v_2)$ can only affect the $q.items$ array at indexes larger than i_1, $(*)$ is an invariant.

Given this invariant, however, it is impossible for d to return v_2, as in its loop it will necessarily first have encountered v_1.

5 Automation

As can be seen from our previous informal argument, establishing absence of VFresh and VRepet violations was relatively straightforward, whereas proving POrd was somewhat more involved. Therefore, in this section, we will focus on automating the proof of the third property, POrd. Towards the end of the section, we will discuss the automatic verification of the absence of VWit violations for queue implementations, where `deq` may return `NULL`.

Prophetic Instrumentation of Dequeues. Our proof technique relies heavily on instrumenting the $\texttt{deq}()$ function with a prophecy variable 'guessing' the value that will be returned when calling it. Essentially, we construct a method, $\texttt{deq}(v)$, such that the set of traces of $\bigsqcup_{x \in \mathbb{N} \cup \{\texttt{NULL}\}} \texttt{deq}(x)$ is equal to the set of traces of $\texttt{deq}()$, where $\sqcup$ stands for non-deterministic choice. Figure 2 shows the resulting automatically-generated instrumented definition of $\texttt{deq}(v)$ for the HW queue.

Our implementation of the instrumentation performs a sequence of simple rewrites, each of which does not affect the set of traces produced:

$$
\begin{array}{rl}
\textbf{return } E & \rightsquigarrow \mathsf{assume}(v = E); \textbf{return } E \\
\textbf{if } B \textbf{ then } C \textbf{ else } C' & \rightsquigarrow (\mathsf{assume}(B); C) \sqcup (\mathsf{assume}(\neg B); C') \\
C; \mathsf{assume}(B) & \rightsquigarrow \mathsf{assume}(B); C \qquad \text{provided } fv(B) \subseteq \mathit{Locals} \setminus \mathit{writes}(C) \\
C; (C_1 \sqcup C_2) & \leftrightsquigarrow (C; C_1) \sqcup (C; C_2) \\
(C_1 \sqcup C_2); C & \leftrightsquigarrow (C_1; C) \sqcup (C_2; C)
\end{array}
$$

In general, the goal of applying these rewrite rules is to bring the introduced $\mathsf{assume}(v = E)$ statements as early as possible without unduly duplicating code.

Proving Absence of VOrd *Violations.* It turns out that our automated technique for proving POrd also establishes absence of VFresh violations as a side-effect. We reduce the

problem of proving absence of VFresh and VOrd violations to the problem of checking non-termination of non-deterministic programs with an unbounded number of threads. The reduction exploits the instrumented $\texttt{deq}(v)$ definition: $\texttt{deq}()$ cannot return a result x in an execution precisely if $\texttt{deq}(x)$ cannot terminate in that same execution.

Theorem 3. *A completable queue implementation has no* VFresh *and* VOrd *violations iff for all* $n \in \mathbb{N}$ *and forall* v_1 *and* v_2 *such that* $v_1 \neq v_2$*, the program*[1]

$$Prg \stackrel{\text{def}}{=} b \leftarrow \mathsf{false};\ (\texttt{deq}(v_2) \parallel \overbrace{C \parallel \ldots \parallel C}^{n \text{ times}})$$

does not terminate, where

$$C \stackrel{\text{def}}{=} (\texttt{enq}(v_1); b \leftarrow \mathsf{true}) \sqcup (\mathsf{assume}(b); \texttt{enq}(v_2)) \sqcup \bigsqcup_{x \neq v_2} \texttt{enq}(x) \sqcup \bigsqcup_{x \neq v_1} \texttt{deq}(x)\,.$$

Proof. ($\Rightarrow$) We argue by contradiction. Consider a terminating history c of Prg. If $\texttt{enq}(v_2)$ is not invoked in c, then as there are no VFresh violations, we know that no $\texttt{deq}()$ in c can return v_2, contradicting our assumption that c is a terminating history of Prg. Otherwise, if $\texttt{enq}(v_2)$ is invoked in c, then at some earlier point $\mathsf{assume}(b)$ was executed, and since initially b was set to false, this means that $b \leftarrow \mathsf{true}$ was executed and therefore $\texttt{enq}(v_1) \prec_c \texttt{enq}(v_2)$. Consequently, from POrd, if there is $\texttt{deq}()$ in c returns v_2, there must be a $\texttt{deq}()$ in c that can be completed to return v_1, contradicting our assumption that c is a terminating history of Prg.

($\Leftarrow$) We have two properties to prove. For VFresh, it suffices to consider the restricted parallel context that never chooses to execute the first two of the non-deterministic choices. In this restricted context, namely one that never enqueues v_2, $\texttt{deq}(v_2)$ does not terminate, and so $\texttt{deq}()$ cannot return v_2. For VOrd, consider a history in which every $\texttt{enq}(v_2)$ happens after some enqueue of a different value, say $\texttt{enq}(v_1)$, and in which there is no $\texttt{deq}(v_1)$. Such a history can easily be produced by the unbounded parallel composition of C, and so $\texttt{deq}(v_2)$ also does not terminate, as required. □

To prove non-termination, we essentially prove the partial-correctness Hoare triple, $\{\mathsf{true}\}\ Prg\ \{\mathsf{false}\}$. Given a sound program logic, the only way for such a triple to hold is for the program to always diverge.

Implementation within CAVE. To prove such triples, we have mildly adapted the implementation of CAVE [15], a sound but incomplete thread-modular concurrent program verifier that can handle dynamically allocated linked list data structures, fine-grained concurrency. The tool takes as its input a program consisting of some initialization code and a number of concurrent methods, which are all executed in parallel an unbounded

[1] For simplicity, we assume that the methods cannot distinguish the thread in which they are running (i.e., they do not use thread-local storage or thread identifiers). Handling thread identifiers properly is not difficult: we have to record a set of thread identifiers that are not currently in use. Before each method invocation, we have to atomically pick and remove an identifier from that set, and on returning from the method, we have to add the current identifier back the set of unused identifiers.

number of times each. When successful, it produces a proof in RGSep that the program has no memory errors and none of its assertions are violated at runtime. Internally, it performs RGSep action inference [16] with a rich shape-value abstract domain [14] that can remember invariants of the form that value v_1 is inside a linked list. CAVE also has a way of proving linearizability by a brute-force search for linearization points (see [15] for details), but this is not applicable to the HW queue and therefore irrelevant for our purposes.

The main modifications we had to perform to the tool were: (1) to add code that instruments deq() methods with a prophecy argument guessing its return value, thereby generating $\texttt{deq}(v)$; (2) to improve the abstraction function so that it can remember properties of the form $v_2 \notin X$, which are needed to express the $(*)$ invariant of the proof in §4; and (3) to add some glue code that constructs the Prg verification condition and runs the underlying prover to verify it.

As CAVE does not support arrays (it only supports linked lists), we gave the tool a linked-list version of the HW queue, for which it successfully verified that there are no VFresh and VOrd violations.

Showing Absence of VWit *Violations.* Here, we have to show that any dequeue event cannot return empty if it never goes through a state where the queue is logically empty. This in turn means that we have to express non-emptiness using only the actions of the history (and not referring to the linearization point or the gluing invariant which relates the concrete states of the implementation to the abstract states of the queue). For the following let us fix a (complete) concurrent history c and a dequeue of interest d which returns NULL and does not precede any other event in c.

Let c' be some prefix of c and let $e \in Enq(c')$ be a complete enqueue event in c'. We will call e *alive* after c' if there is no completion of c' in which the dequeue event $\texttt{deq}(Val_{c'}(e))$ occurs. Let d_i denote the dequeue event which removes the element inserted by the enqueue event e_i; that is, $d_i = \texttt{deq}(Val_c(e_i))$. A sequence $e_0 e_1 \dots e_n$ of enqueue events in $Enq(c)$ is *covering* for d in c if the following holds:

- e_0 is alive at c' where c' is the maximal prefix of c such that $d \notin Deq(c')$.
- For all $i \in [1, n]$, e_i starts before d completes.
- For all $i \in [1, n]$, we have $e_i \prec_c d_{i-1}$.
- e_n is alive at c.

Note that all d_i must exist by the third condition and that d_n does not exist by the last condition. Then, the sequence is covering for d if d_0 does not start before d starts, and every enqueue event e_i completes before the dequeue event d_{i-1} starts. Intuitively, this means that at every state visited during the execution of d, the queue contains at least one element. The property corresponding to the last violation (VWit) then becomes the following:

(PWit)**.** A dequeue event d cannot return NULL if there is a covering for d.

We will actually re-state the same property in a simpler way by making the following observation.

Proposition 5. *There is a covering for d in c iff at every prefix c' of c such that d is running, there is at least one alive enqueue event.*

Then, we can alternatively state PWit as follows:

(PWit′). A dequeue event d cannot return `NULL` if for every prefix c' at which d is pending there exists an alive enqueue event.

Note that POrd can also be stated in terms of alive enqueue events.

(POrd′). For any enqueue events e_1 and e_2 with $e_1 \prec_c e_2$ and $Val_c(e_1) \neq Val_c(e_2)$, a dequeue event cannot return $Val_c(e_2)$ if e_1 is alive at c.

6 Related Work

Linearizability was first introduced by Herlihy and Wing [8], who also presented the HW queue as an example whose linearizability cannot be proved by a simple forward simulation where each method performs its effects instantaneously at some point during its execution. The problem is, as we have seen, that neither of E_1 or E_2 can be given as the (unique) linearization point of `enq` events, because the way in which two concurrent enqueues are ordered may depend on not-yet-completed concurrent `deq` events. In other words, one cannot simply define a mapping from the concrete HW queue states to the queue specification states. Nevertheless, Herlihy and Wing do not dismiss the linearization point technique completely, as we do, but instead construct a proof where they map concrete states to non-empty sets of specification states.

This mapping of concrete states to non-empty sets of abstract states is closely related to the method of *backward simulations*, employed by a number of manual proof efforts [3, 5, 13], and which Schellhorn et al. [13] recently showed to be a complete proof method for verifying linearizability. Similar to forward simulation proofs, backward simulation proofs, are monolithic in the sense that they prove linearizability directly by one big proof. Sadly, they are also not very intuitive and as a result often difficult to come up with. For instance, although the definition of their backward simulation relation for the HW queue is four lines long, Schellhorn et al. [13] devote two full pages to explain it.

As a result, most work on automatically verifying linearizability (e.g. [1, 2, 14, 15]) has relied on the simpler technique of forward simulations, even though it is known to be incomplete. The programmer is typically required to annotate each method with its linearization points and then the verifier uses some kind of shape analysis that automatically constructs the simulation relation. This approach seems to work well for simple concurrent algorithms such as the Treiber stack and the Michael and Scott queues, where finding the linearization points may be automated by brute-force search [15], but cannot handle more challenging examples such as the ones mentioned in the introduction.

Among this line of work, the most closely related one to this paper is the recent work by Abdulla et al. [1], who verify linearizability of stack and queue algorithms using observer automata that report specification violations such as our VOrd. Their approach, however, still requires users to annotate methods with linearization points,

because checker automata are synchronized with the linearization points of the implementation.

We would also like to point out that the use of forward simulations is not limited to automated verifications of linearizability. Several manual verification efforts have also used forward simulations (e.g. [3, 4]).

To the best of our knowledge, there exist only two earlier published proofs of the HW queue: (1) the original pencil-and-paper proof by Herlihy and Wing [8], and (2) a mechanized backward simulation proof by Schellhorn et al. [13].

Both proofs are manually constructed. In comparison, our new proof is simpler, more modular, and largely automatically generated.[2] This is largely due to the fact that we have decomposed the goal of proving linearizability into proving four simpler properties, which can be proved independently. This may allow one to adapt the HW queue algorithm, e.g. by checking emptiness of the queue and allowing `deq` to return `NULL`, and affecting only the proof of absence of VWit violations without affecting the correctness arguments of the other properties.

Our violation conditions are arguably closer to what programmers have in mind when discussing concurrent data structures. Informal specifications written by programmers and bug reports do not mention that some method is not linearizable, but rather things like that values were dequeued in the wrong order.

7 Conclusion

We have presented a new method for checking linearizability of concurrent queues. Instead of searching for the linearization points and doing a monolithic simulation proof, we verify four simple properties whose conjunction is equivalent to linearizability with respect to the atomic queue specification. By decomposing linearizability proofs in this way, we obtained a simpler correctness proof of the Herlihy and Wing queue [8], and one which can be produced automatically.

We believe that our new property-oriented approach to linearizability proofs will be equally applicable to other kinds of concurrent shared data structures, such as stacks, sets, and maps. In the future, we would like to build tools that will automate this kind of reasoning for such data structures.

Acknowledgments. We would like to thank the reviewers for their feedback. The research was supported by the EC FET FP7 project ADVENT, by the Austrian Science Fund NFN RISE (Rigorous Systems Engineering) and by the ERC Advanced Grant QUAREM (Quantitative Reactive Modeling).

References

1. Abdulla, P.A., Haziza, F., Holík, L., Jonsson, B., Rezine, A.: An integrated specification and verification technique for highly concurrent data structures. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 324–338. Springer, Heidelberg (2013)

[2] We say 'largely' because we have not yet automated the verification of the absence of VRepet violations, which requires a simple counting argument, nor the (admittedly trivial) proof that the HW queue is purely-blocking. We intend to implement these in the near future.

2. Amit, D., Rinetzky, N., Reps, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 477–490. Springer, Heidelberg (2007)
3. Colvin, R., Doherty, S., Groves, L.: Verifying concurrent data structures by simulation. ENTCS 137(2), 93–110 (2005)
4. Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying linearisability with potential linearisation points. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 323–337. Springer, Heidelberg (2011)
5. Doherty, S., Moir, M.: Nonblocking algorithms and backward simulation. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 274–288. Springer, Heidelberg (2009)
6. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: SPAA 2010, pp. 355–364. ACM (2010)
7. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc. (2008)
8. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
9. Hoffman, M., Shalev, O., Shavit, N.: The baskets queue. In: Tovar, E., Tsigas, P., Fouchal, H. (eds.) OPODIS 2007. LNCS, vol. 4878, pp. 401–414. Springer, Heidelberg (2007)
10. Ladan-Mozes, E., Shavit, N.: An optimistic approach to lock-free FIFO queues. In: Guerraoui, R. (ed.) DISC 2004. LNCS, vol. 3274, pp. 117–131. Springer, Heidelberg (2004)
11. Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model checking linearizability via refinement. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 321–337. Springer, Heidelberg (2009)
12. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free FIFO queues. In: SPAA 2005, pp. 253–262. ACM (2005)
13. Schellhorn, G., Wehrheim, H., Derrick, J.: How to prove algorithms linearisable. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 243–259. Springer, Heidelberg (2012)
14. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 335–348. Springer, Heidelberg (2009)
15. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010)
16. Vafeiadis, V.: RGSep action inference. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 345–361. Springer, Heidelberg (2010)

Causality-Based Verification of Multi-threaded Programs*

Andrey Kupriyanov and Bernd Finkbeiner

Universität des Saarlandes, Saarbrücken, Germany

Abstract. We present a new model checking procedure for concurrent systems against safety properties such as data races or atomicity violations. Our analysis sidesteps the state space explosion problem by inferring causal dependencies for concurrent traces instead of searching over a space of reachable states, and can be understood as an interplay between local trace inference and termination analysis based on causal loops. Local trace inference introduces new actions anywhere in the trace if they causally follow from the context. Our procedure terminates if we either find a complete error trace or the whole space of potential error traces is covered by causal loops. The causality-based verification of multi-threaded programs can be dramatically faster than the standard state space traversal. In particular, we show that the complexity of verifying multi-threaded programs with locks reduces from exponential to polynomial.

1 Introduction

Causality, the relationship between two events where the first event is recognized as a necessary requirement for the occurrence of the second, is a key concept in our understanding of complex computer systems. Processes in a concurrent system proceed independently until they establish causal dependence through synchronization. Modeling formalisms like Petri nets [10] explicitly capture the causal dependence between transitions through the flow of tokens.

Not surprisingly, causality has also proven useful in the automatic verification of concurrent systems. Petri net unfoldings [4], for example, avoid a total temporal ordering of the events and instead unwind the causality relation. In partial order reduction [5], causally independent events are forced into a fixed temporal order. Traditionally, however, the role of causality in automatic verification has always been secondary compared to the state-based reachability analysis: in Petri net unfoldings, we unwind the causality relation forward until we are certain that no more reachable markings can be found; in partial order reduction, we avoid the exploration of computation paths that lead to the same states that have already been seen on some other path with a different ordering of the causally independent events.

* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, `www.avacs.org`).

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 257–272, 2013.

In this paper, we upgrade the role of causality in automatic verification to that of a first-class citizen. The approach is based on the observation that reaching an error state often causally depends on a small number of certain key events, which, in a correct system, contradict each other. For example, a violation of mutual exclusion between two processes requires that previously both processes have entered the critical section and, during one of these events, the other process was already in the critical section. Intuitively, our verification procedure identifies such necessary steps on a trace from initial to error states and then either completes the partial trace into a full error trace or proves that no such completion exists.

In our algorithm, we capture the causal dependencies as Mazurkiewicz-style concurrent traces. Starting with a default initial trace, which only captures the initial and error states, we capture, step by step, more dependencies by applying special graph transformations, which we call *causal transitions*: the causal transitions include, for example, the *necessary action* transition, which uses Craig interpolation to find a necessary intermediate action. A full exploration of the causal dependencies leads to an in general infinite tree, which we call the *causal trace unwinding*. If all branches of the causal trace unwinding are either contradictory (meaning that the causal requirements of reaching the error cannot be satisfied) or infinite (meaning that no finite number of actions suffices to reach the error), we can conclude that the error is, in fact, unreachable. The verification algorithm builds finite prefixes of the causal trace unwinding and terminates as soon as the two conditions can be established for the full tree.

The causality-based verification of multi-threaded programs can be dramatically faster than the standard state space traversal. In the paper, we demonstrate this effect for multi-threaded programs with locks. It turns out that our algorithm verifies the most general class of these programs in polynomial time. This answers an open question originally posed by Alexander Malkis [7].

The remainder of the paper is structured as follows. After a brief discussion of related work, we consider a motivating example from the class of multi-threaded programs with locks in Section 2. We discuss the necessary preliminaries in Section 3 and define the central structure of our approach, causal trace unwinding, in Section 4. The causality-based verification algorithm, which allows us to explore only a finite prefix of the unwinding, is described in Section 5. Finally, in Section 6 we show how our verification algorithm settles the open question regarding the verification of multi-threaded programs with locks, reducing the complexity from exponential to polynomial.

Related Work. Causality-based verification can be understood as a generalization of standard model checking [1], because the next-state relation usually explored in model checking captures the causal dependencies between successor states: a trace from some arbitrary non-error state to an error state can only exist if there is a trace from one of the state's successors. However, it is usually easy to obtain additional elements of the causality relation, for example based on a cheap analysis of the control flow graph. Such additional elements are exploited

by our procedure, but not by standard model checking based on a forward or backward traversal of the state space.

Our method is related to approaches based on partial orders, such as partial order reduction [5], Mazurkiewicz traces [9], and Petri net theory [10]. Similar to causality-based verification, these approaches exploit the independence that results from the combination of separate processes. Unlike these approaches, we do not require, however, that the system is given a-priori as a partial order. Our rules extract causal dependencies from the system description and use such dependencies to contradict the existence of an error trace. Finally, causality-based verification is a tableau-based decision procedure, related to the tableau-based approaches for modal and temporal logics [8].

2 Motivating Example

As a motivating example we consider the class of multithreaded programs with critical sections protected by shared lock variables, which is described in [7]. A program in this class consists of n threads, executing in the interleaved fashion, and m shared boolean lock variables. Each thread contains some finite number of critical sections, protected by the "*acquire* lck_i" and "*release* lck_i" statements for one of the lock variables. Critical sections may be arbitrarily nested or intersected, and a thread may have an arbitrary control structure via the use of "*if* (φ) *goto* j" statements, where φ is any formula over the shared variables. The only restriction for a correct program is that the control flow may enter one of the critical sections for the lck_i variable only via the "*acquire* lck_i" statement, and may exit it either by jumping to another critical section for the same lock variable, or by executing the "*release* lck_i" statement. The syntax and semantics of such programs are shown in the left part of Figure 1; they can be used to analyze systems with built-in "test-and-set" primitive. In the following we consider the example program depicted on the right of Figure 1; we want to verify that threads 1 and 2 cannot be simultaneously at their critical sections 2, protected by lock l_1.

Our algorithm operates on a causal trace unwinding, where vertices are labeled with abstract traces. In Figure 2 we depict the unwinding in the center, and the labels of its vertices on the left or on the right from a corresponding vertex (we do not draw some trace edges, which follow from transitivity).

Step 1: We start with an unwinding, containing a single vertex 1, and labeled with the abstract trace representing all concrete traces from initial state to error state (initial action i is labeled with $pc'_1 = 1 \wedge pc'_2 = 1 \wedge pc'_3 = 1$, and error action e is labeled with $pc_1 = 2 \wedge pc_2 = 2$).

Step 2: We check whether the abstract trace of vertex 1 is concretizable. It is not, for example because of the of the conflict $pc'_1 = 1 \not\downarrow pc_1 = 2$ between actions i and e. We conclude that in between of initial and error actions a *necessary action*, characterized by the transition predicate $pc_1 \neq 2 \wedge pc'_1 = 2$, should happen. There is only one system transition, a_1, satisfying this predicate, and we introduce new vertex 2 in the unwinding, labeled with the abstract trace

Syntax	Semantics
acquire l_i	$l_i = 0 \wedge l_i' = 1 \wedge pc' = pc + 1$
release l_i	$l_i' = 0 \wedge pc' = pc + 1$
if (φ) *goto* j	$(\varphi \wedge pc' = j) \vee (\neg\varphi \wedge pc' = pc+1)$

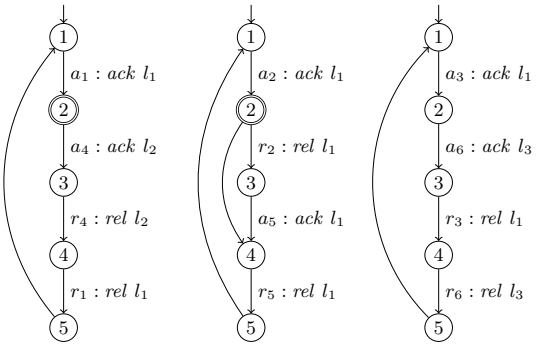


Fig. 1. General class of multithreaded programs with binary locks. *Left:* Syntax and semantics. *Right:* Example system consisting of 3 threads with critical sections over 3 lock variables. Initial state vector is $(1, 1, 1)$, and error state vector is $(2, 2, \_)$.

where transition a_1 is inserted. We require that thread 1 does not leave location 2, and mark the edge $a_1 \to e$ with the predicate $pc_1 = 2$.

Step 3: is similar to step 2: there is a conflict $pc_2' = 1 \not\downarrow pc_2 = 2$, and a single necessary action, a_2, satisfying the transition predicate $pc_2 \neq 2 \wedge pc_2' = 2$. We introduce new vertex 3, where actions a_1 and a_2 are concurrent: they are both necessary, but the order of their occurrence is unspecified.

Step 4: We try to linearize the abstract trace from vertex 3; it contains two concurrent actions, and we choose an arbitrary order between them, for example a_1 before a_2. The linear trace contains the conflict $l' = 1 \not\downarrow l = 0$ (a_1 has the postcondition $l = 1$, while a_2 has the precondition $l = 0$). But, because the order between a_1 and a_2 is not dictated by the abstract trace, we make an *order split*, considering both alternatives (vertices 4 and 5).

Step 5: We consider only vertex 5 from the previous step; vertex 4 is analyzed analogously. The conflict $l' = 1 \not\downarrow l = 0$ between a_1 and a_2 is now dictated by the trace; so a new necessary action, satisfying the predicate $l \neq 0 \wedge l' = 0$ should be inserted. We instantiate this abstract action with all concrete actions, satisfying the predicate, namely r_1, r_2, r_3, and r_5. Here, for the picture clarity, we show only vertices 6 (with r_1) and 7 (with r_2).

Step 6: Consider first the trace of vertex 6: it contains a contradiction, namely action r_1 (labeled with $pc_1 = 4 \wedge pc_1' = 5 \wedge l' = 0$) lies in the scope of the edge $a_1 \to e$ (labeled with $pc_1 = 2$), and their labels are unsatisfiable together. Thus, we close this branch as contradictory.

Step 7: For each leaf vertex we try to find whether a "similar" concurrent trace was already encountered before. For the case of vertex 7, its label is, indeed, similar to the label of vertex 1: we can find a mapping from all nodes and edges of the latter to the nodes and edges of the former. More precisely, we can map node i to node i, and node e to node r_2. Moreover, the label of node r_2, which equals to $pc_2 = 2 \wedge pc_2' = 3 \wedge l' = 0 \wedge pc_1 = 2$ due to the restriction from the

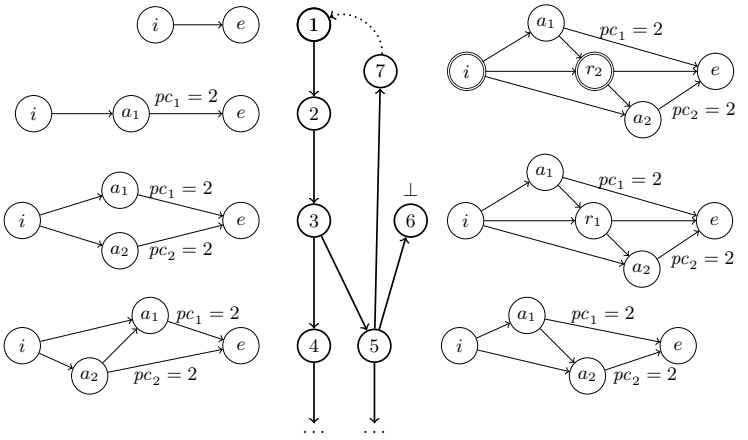


Fig. 2. First steps of the causal trace unwinding for the example multi-threaded program with binary locks

edge $a_1 \to e$, implies the label of node e. Thus, the trace of vertex 7 is more restrictive than the trace of vertex 1, and we can *cover* vertex 7 by vertex 1. Also, there is node a_2 on the right of r_2, which the covering "forgets". The path $1 \to 2 \to 3 \to 5 \to 7 \to 1$ in the unwinding constitutes a *causal loop*: as long as we follow the loop, we keep introducing new and new actions a_2 on the right.

Continuing the process, we would find out that all leaf nodes in the trace unwinding are either contradictory, as in step 6, or are covered by such causal loops as in step 7. This implies that our system is correct, because any possible error trace would have infinite length. We will return to the verification of multi-threaded programs with locks in Section 6.

3 Preliminaries

Transition Systems. We consider concurrent systems described in some first-order assertion language. For a set of variables $\mathcal{V}$, we denote by $\Phi(\mathcal{V})$ the set of first-order formulas over $\mathcal{V}$. For each variable $x \in \mathcal{V}$ we define a primed variable $x' \in \mathcal{V}'$, which denotes the value of x in the next state. We call formulas from the sets $\Phi(\mathcal{V})$ and $\Phi(\mathcal{V} \cup \mathcal{V}')$ *state predicates* and *transition predicates*, respectively.

A *transition system* is a tuple $\mathcal{S} = \langle \mathcal{V}, T, init, error \rangle$ where $\mathcal{V}$ is a finite set of system variables; $T \subseteq \Phi(\mathcal{V} \cup \mathcal{V}')$ is a finite set of system transitions; $init \in \Phi(\mathcal{V})$ and $error \in \Phi(\mathcal{V})$ are state predicates, characterizing initial and error states.

A *state* of $\mathcal{S}$ is a valuation of system variables $\mathcal{V}$. We call an alternating sequence of states and transitions $s_0, t_1, s_1, t_2, \ldots, t_n, s_n$ a *trace*, if $init(s_0)$ holds, and for all $1 \leq i \leq n$, $t_i(s_{i-1}, s_i)$ holds. We call a trace $s_0, t_1, s_1, t_2, \ldots, t_n, s_n$ an *error trace* if it ends in some error state, i.e. the predicate $error(s_n)$ holds. We say that the system is *safe* if there does not exist any error trace for that system; otherwise the system is *unsafe*. For a system $\mathcal{S}$ we denote the set of its traces by $\mathcal{L}(\mathcal{S})$, and the set of its error traces by $\mathcal{L}_e(\mathcal{S}) \subseteq \mathcal{L}(\mathcal{S})$.

Transition systems are well suited for the representation of multi-threaded programs with interleaving semantics: in this case the set of system transitions is simply a union of transitions of individual processes.

Graph Transformations. We follow [2,3] and use the so-called *single-pushout (SPO)* and *double-pushout (DPO)* approaches to describe graph transformations. All graph transformations that we use are non-erasing and lie at the intersection of both approaches; the definitions below are adapted from [2].

A *graph* is a tuple $G = \langle N, E \rangle$, where N is a set of nodes, and $E \subseteq N \times N$ is a set of edges. The source and target functions $s, t : E \to N$ map each edge to its first and second component, respectively.

Given two graphs $G = \langle N, E \rangle$ and $G' = \langle N', E' \rangle$, a *graph morphism* $f : G \to G'$ is a pair $f = \langle f_N : N \to N', f_E : E \to E' \rangle$ of functions, preserving sources and targets: $f_N \circ t = t' \circ f_E$, and $f_N \circ s = s' \circ f_E$.

For our purposes, a *graph production* $p : (L \xrightarrow{r} R)$ is an injective graph morphism $r : L \to R$. The graphs L and R are called the *left-hand side* and the *right-hand side* of p, respectively. A given production $p : (L \xrightarrow{r} R)$ can be applied to a graph G if there is an occurrence of L in G, i.e. an injective graph morphism $m : L \to G$, called a *match*. In this case the resulting graph H can be obtained from G by adding all elements of R with no pre-image in L. The application of a production p to a graph G with a match m is called a *direct derivation*; we will denote it interchangeably with $G \xRightarrow{p,m} H$ and $H = p^m(G)$.

4 Causal Trace Unwindings

4.1 Concurrent Traces

We follow the theory of Mazurkiewicz traces, and define concurrent traces through their *dependence graphs*. A *concurrent trace* is a labeled, directed, acyclic graph $A = \langle N, E, \nu, \eta \rangle$, where $\langle N, E \rangle$ is a graph with nodes N, called *actions*, and edges E; $\nu : N \to \Phi(V \cup V')$, $\eta : E \to \Phi(V \cup V')$ are labelings of nodes and edges with transition predicates. We denote the set of concurrent traces by $\mathbb{A}$.

A concurrent trace describes a set of system traces. For a particular concurrent trace its actions specify which transitions should necessarily occur in a system trace, while its edges represent the (partial) ordering between such transitions and constraint the intermediate ones.

Trace Language. For a transition system $\mathcal{S} = \langle \mathcal{V}, T, init, error \rangle$, the *language* of a concurrent trace $A = \langle N, E, \nu, \eta \rangle$ is defined as a set $\mathcal{L}(A)$ of system traces such that for each trace $s_0, t_1, s_1, t_2, \ldots, t_n, s_n \in \mathcal{L}(A)$ there exists an injective mapping $\sigma : N \to \{t_1, \ldots, t_n\}$ such that:

1. for each action $a \in N$ and $t_i = \sigma(a)$ the formula $\nu(a)\big(s_{i-1}, s_i\big)$ holds.
2. for each edge $e = (a_1, a_2) \in E$, and $t_i = \sigma(a_1)$, $t_j = \sigma(a_2)$, we have that a) $i < j$, and b) for all $i < k < j$, the formula $\eta(e)\big(s_{k-1}, s_k\big)$ holds.

We call a concurrent trace $A = \langle N, E, \nu, \eta \rangle$ *contradictory* if some of its actions is labeled with an unsatisfiable predicate, i.e. if there exists $n \in N$ such that $\nu(n)$ implies $\bot$. Obviously, the language of such a trace is empty.

Trace Inclusion. For any two concurrent traces $A = \langle N, E, \nu, \eta \rangle$ and $A' = \langle N', E', \nu', \eta' \rangle$ we define the *trace inclusion* relation $\subseteq$ as follows: $A \subseteq A'$ iff

1. there exists a graph morphism $\lambda = \langle \lambda_N : N' \to N, \lambda_E : E' \to E \rangle$.
2. for all $n' \in N' \,.\, \nu(\lambda_N(n')) \implies \nu'(n')$.
3. for all $e' \in E' \,.\, \eta(\lambda_E(e')) \implies \eta'(e')$.

Proposition 1. *if* $A \subseteq A'$ *then* $\mathcal{L}(A) \subseteq \mathcal{L}(A')$.

We write $A \subseteq_\lambda A'$, if trace inclusion holds for a particular graph morphism λ. Let λ^N be the image of λ_N: $\lambda^N = \{n \in N \mid (n', n) \in \lambda_N\}$. We call the trace inclusion $A \subseteq_\lambda A'$ *left-forgetful* (resp. *right-forgetful*), if for all $n \in \lambda^N$ there exists $n_\times \in N \setminus \lambda^N$ such that $(n_\times, n) \in E$ (resp. $(n, n_\times) \in E$). We call the trace inclusion *forgetful* if it is either left- or right-forgetful. Intuitively, when $A \subseteq_\lambda A'$ is a forgetful trace inclusion, we "forget" some action on the left or on the right when moving from A to A'.

Our intention is to find causal consequences from the information about error traces, represented in the form of concurrent traces. For that purpose we start with a single concurrent trace, containing two actions: initial action i, marked with $init'$, and error action e, marked with *error*, connected with an unrestricted edge. The marking ensures that all possible error traces are preserved.

Initial Abstraction. For a transition system $\mathcal{S} = \langle \mathcal{V}, T, init, error \rangle$ we define $\mathit{InitialAbstraction}(\mathcal{S})$ as a concurrent trace $A = \langle N, E, \nu, \eta \rangle$, where $N = \{i, e\}$, $E = \{(i, e)\}$, $\nu = \{(i, init'), (e, error)\}$, $\eta = \{((i, e), true)\}$.

Proposition 2. $\mathcal{L}_e(\mathcal{S}) \subseteq \mathcal{L}(\mathit{InitialAbstraction}(\mathcal{S}))$.

Trace Productions. We lift graph morphisms to traces with the same meaning (mappings for nodes and edges of one trace to those of another), and call them *trace morphisms*. We generalize graph productions to concurrent traces: a *trace production* $\tau : (L \xrightarrow{r} R)$, where L, R are concurrent traces and r is a trace morphism, describes a transformation of trace L into trace R. The graphical part is transformed by the corresponding graph production, and labels are transformed by the operations of boolean algebra. Formally trace productions can be described as graph productions on *attributed graphs*; for details we refer the interested reader to [3], pp. 284-288. In the following we denote the set of trace productions by Π.

4.2 Causal Transitions

Starting from the initial abstraction, we find, step by step, further causal dependencies. For this purpose we introduce in the following special graph productions, which we call *causal transitions*:

Causal Transition. For a given transition system $\mathcal{S}$, a *causal transition* $\tau : \{\tau_1, \ldots, \tau_n\}$ is a set of trace productions $\tau_i : (L \xrightarrow{r_i} R_i)$, where all productions share the same left-hand side L; we will denote L by $\tau_\triangleleft$, and call transition *premise*. We say that causal transition τ is *sound* if the condition below holds:

$$\forall A \in \mathbb{A} \,.\, A \subseteq_m \tau_\triangleleft \implies \mathcal{L}(A) \subseteq \bigcup_{\tau_i \in \tau} \mathcal{L}\big(\tau_i^m(A)\big)$$

The above condition says that if the transition premise $\tau_{\lhd}$ can be matched to some concurrent trace A, then the application of the transition should preserve all possible concrete traces, contained in A. Please note, that causal transitions can be interpreted both operationally (as transformations of concurrent traces), and logically (as language inclusion); we employ both interpretations. In the following we denote the set of causal transitions by Δ.

Below we describe some examples of causal transitions, shown in Figure 3.

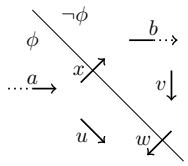
Order Split (Figure 3a). The *order split* causal transition considers alternative interleavings of two previously concurrent events.

Action Split (Figure 3b). The *action split* causal transition, given some action a in the trace, and a transition predicate ψ, considers two alternatives: either a satisfies ψ or not.

Transitivity (Figure 3c). The *transitivity* causal transition, given two sequential edges $a \to b$ and $b \to c$, allows to introduce edge $a \to c$, which follows from transitivity, and label it with the disjunction of the constraints in its scope.

Necessary Action (Figure 3d). The *necessary action* causal transition, given two ordered actions a and b in a concurrent trace, and a transition predicate ϕ, such that the label of a implies ϕ', and the label of b implies $\neg\phi$, i.e. there is a contradiction between these actions (a "ends" in the region ϕ, while b "starts" in the region $\neg\phi$), introduces a new "bridging" action x in between. The predicate ϕ may be obtained by Craig interpolation between the labels of a and b. The application condition for this causal transition ensures that there is no other action y in the trace already, that could play the role of x.

Figure 3d shows three causal transitions, which have the same left-hand-side L, but different right-hand sides R, R_{first}, and R_{last} (separated by vertical bars in the figure). In R we simply insert action x in the concurrent trace. In R_{first} (resp. R_{last}) we require, additionally, that action x is the first (resp. last) action, that crosses the boundary between ϕ and $\neg\phi$ (see picture on the right). This can be achieved by marking the corresponding edge with the predicate $\neg(\phi \wedge \neg\phi') = \neg\phi \vee \phi'$; but we can easily strengthen this requirement. Indeed, suppose we want action x to be the last in the sequence of possible necessary actions; the only actions allowed by the above predicate are of the type u ($\phi \wedge \phi'$), v ($\neg\phi \wedge \neg\phi'$), or w ($\neg\phi \wedge \phi'$). But, if an action of type u or w happens, then an action of type x becomes necessary again, and it is not allowed: a contradiction. Thus, we can safely allow only actions of type v to happen, and strengthen the above predicate to $\neg\phi \wedge \neg\phi'$.



Action/Edge Restriction (Figures 3e, 3f). The *action restriction* (resp. *edge restriction*) causal transition allows us to restrict the label of an action (resp. edge), if it happens to be in the scope of some edge (for example, as a result of *order split* application).

Forward/Backward Unrolling (Figures 3g, 3h). The *forward unrolling* (resp. *backward unrolling*) causal transition, given two actions in a concurrent trace, which cannot follow immediately one after another and do not have

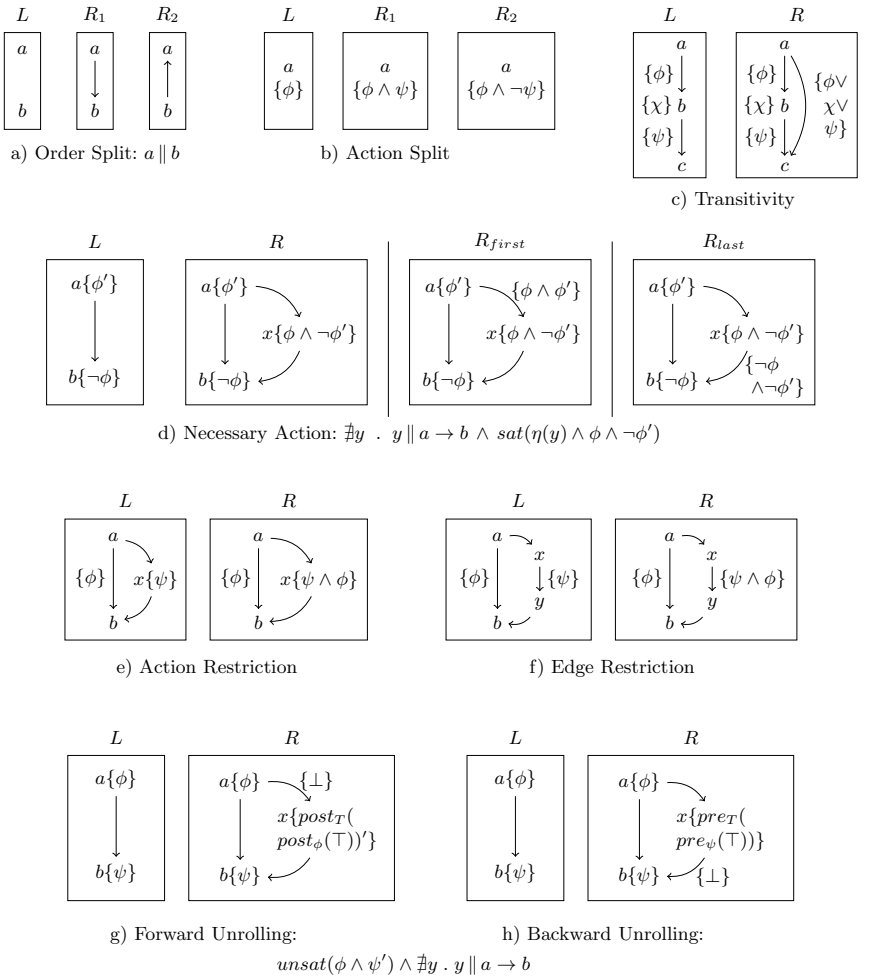


Fig. 3. Examples of causal transitions

any other actions in between, unrolls the transition relation one step forward or backward. Let $\phi_{[\mathcal{V}/\mathcal{V}']}$ denote the substitution of $\mathcal{V}'$ variables in formula ϕ by corresponding variables $\mathcal{V}$. Then $post_\tau(\phi) = \left(\exists_{\mathcal{V}} \ \phi(\mathcal{V}) \wedge \tau(\mathcal{V} \cup \mathcal{V}')\right)_{[\mathcal{V}/\mathcal{V}']}$ is a post-image of ϕ with respect to transition relation τ, while $pre_\tau(\phi) = \exists_{\mathcal{V}'} \ \tau(\mathcal{V} \cup \mathcal{V}') \wedge \phi(\mathcal{V})_{[\mathcal{V}'/\mathcal{V}]}$ is a pre-image of ϕ with respect to τ. For the case of forward unrolling, to calculate the label of the newly introduced action x, we first compute the post-image of the preceding action a, and then the post-image of the result with respect to the whole transition relation T (treated here, by abuse of notation, as a disjunction of all transitions from T). The label $\bot$ on the edge $a \rightarrow x$ ensures that there are no other actions between a and x. Calculations for the case of backward unrolling are done similarly.

Proposition 3. *The defined above causal transitions are sound.*

4.3 Causal Trace Unwindings

Causal Trace Unwinding. For a transition system $\mathcal{S}$, we define a *(causal) trace unwinding* as a tuple $\Upsilon = \langle V, F, \gamma, \delta, \lambda \rangle$, where:

- (V, F) is a directed tree with vertices V, root vertex $v_0 \in V$, and edges F. Vertices are partitioned into internal vertices and leaves: $V = V_N \uplus V_L$, $V_N = \{v \in V \mid \exists (v, v') \in F\}$, $V_L = \{v \in V \mid \nexists (v, v') \in F\}$.
- $\gamma : V \to \mathbb{A}$ is a labeling of vertices with concurrent traces.
- $\delta : F \to \Pi$ is a labeling of edges with trace productions. We require that for all edges with the same source v, the labeling productions have the same left-hand side. Thus, we have an induced labeling of internal vertices $v \in V_N$ with causal transitions: $\delta(v) = \{\delta((v, v')) \mid (v, v') \in F\}$.
- λ is a labeling of internal vertices with trace morphisms: $\forall v \in V_N \,.\, \lambda(v) : \delta(v)_\lhd \to \gamma(v)$.

A trace unwinding is said to be *correct* if it satisfies the following criteria:

1. $\mathit{InitialAbstraction}(\mathcal{S}) \subseteq \gamma(v_0)$.
2. for all internal vertices $v \in V_N$ we have: a) $\delta(v)$ is sound, b) $\gamma(v) \subseteq_{\lambda(v)} \delta(v)_\lhd$, and c) for all $(v, v') \in F$ it holds that $\delta\big((v, v')\big)^{\lambda(v)}\big(\gamma(v)\big) \subseteq \gamma(v')$.

A trace unwinding is a tree, which can be seen as an unwinding of the trace causality relation. The label $\gamma(v)$ of the vertex v represents all possible error traces for that vertex; the first condition above ensures that the root vertex v_0 contains all error traces of the given system. The second condition guarantees the applicability of the causal transition $\delta(v)$ of a vertex v to its label $\gamma(v)$ and full exploration of the causal transition consequences, thus preserving the set of concrete traces. Indeed, we have:

$$\gamma(v) \subseteq_{\lambda(v)} \delta(v)_\lhd \implies \mathcal{L}\big(\gamma(v)\big) \subseteq \bigcup_{(v,v') \in F} \mathcal{L}\big(\delta\big((v, v')\big)^{\lambda(v)}\big(\gamma(v)\big)\big) \subseteq \bigcup_{(v,v') \in F} \mathcal{L}\big(\gamma(v')\big)$$

We call *causal path* a finite or infinite sequence $v_0, v_1, v_2, \ldots$ of vertices, starting from the root v_0, such that for all $i \geq 0$, $(v_i, v_{i+1}) \in F$. We call a causal path *contradictory* if it is finite and ends in a vertex labeled with a contradictory trace. We call a causal path *unbounded* if it is infinite and the number of actions in the labeling of its vertices increases beyond any bound: for any $n \in \mathbb{N}$ there exists $i \geq 0$ such that $|\gamma(v_i)| > n$.

Theorem 1 (Soundness of Trace Unwinding). *If there exists a correct causal trace unwinding for a transition system $\mathcal{S}$, where every causal path is either contradictory or unbounded, then $\mathcal{S}$ is safe.*

5 Causality-Based Verification Algorithm

The causal trace unwinding, described in the preceding section, is easy to construct, but in most cases it will be infinite. In this section we provide an algorithm that explores only a finite prefix of an infinite unwinding and, based on that prefix, establishes the desired properties for the whole unwinding.

The idea behind the finite unwinding prefix is simple: as soon as we encounter a new vertex, labeled with some concurrent trace we have seen before, we would like to cut the unwinding at that vertex and loop back. There are two problems with this simple-minded approach. First, the exact match of one trace to another, like in step 7 of the motivating example, is rarely achievable. We solve this problem by tracking for each vertex the most general trace, sufficient to repeat all causal transitions in the subtree of that vertex; in that way we significantly relax the matching requirement. Second, we should ensure that every infinite path in the unwinding is unbounded; we achieve that by requiring that a forgetful trace inclusion holds between the trace of the leaf vertex and the most general trace of the vertex where the back loop leads to. We call such a finite unwinding prefix *causal trace tableau*, and such back loops *covering*.

Causal Trace Tableau. A *(causal) trace tableau* for a transition system $\mathcal{S}$ is a tuple $\langle \Upsilon, \rightsquigarrow, \alpha, \mu, \sigma \rangle$ where:

- $\Upsilon = \langle V, F, \gamma, \delta, \lambda \rangle$ is a causal trace unwinding.
- $\rightsquigarrow\colon V_L \nrightarrow V_N$ is a partial *covering* function; for $(v, v') \in \rightsquigarrow$ we call v a *covered* vertex, and v' a *covering* vertex.
- $\alpha : V \to \mathbb{A}$ is a labeling of vertices with (abstract) concurrent traces.
- μ and σ are labelings of vertices with trace morphisms: $\mu(v) : \delta(v)_\lhd \to \alpha(v)$ and $\sigma(v) : \alpha(v) \to \gamma(v)$ such that $\sigma(v) \circ \mu(v) = \lambda(v)$.

We call a trace tableau $\langle \Upsilon, \rightsquigarrow, \alpha, \mu, \sigma \rangle$ *complete* if all its leaf vertices are either contradictory or covered. We call it *correct* if Υ is correct and, additionally:

1. for all vertices $v \in V$ we have $\gamma(v) \subseteq_{\sigma(v)} \alpha(v) \subseteq_{\mu(v)} \delta(v)_\lhd$.
2. for all $(v, v') \in F$ we have $\delta\big((v, v')\big)^{\mu(v)}\big(\alpha(v)\big) \subseteq \alpha(v')$.
3. for all $(v, v') \in \rightsquigarrow$ we have that $\alpha(v) \subseteq_{\mu(v)} \alpha(v')$ is a forgetful trace inclusion.
4. for all $v \in V_L$ such that $\gamma(v)$ is contradictory, $\alpha(v)$ is also contradictory.

Theorem 2 (Soundness of Trace Tableau). *If there exists a correct and complete causal trace tableau for a transition system $\mathcal{S}$, then $\mathcal{S}$ is safe.*

Theorem 3 (Completeness of Trace Tableau). *If a transition system $\mathcal{S}$ with finite-state quotient is safe, then there exists a correct and complete causal trace tableau for $\mathcal{S}$.*

Our causality-based verification algorithm (see Algorithm 1), operates on the trace tableau defined above. Each vertex v in the tableau is labeled with two concurrent traces: a *concrete* trace $\gamma(v)$, and an *abstract* trace $\alpha(v)$; we always have that $\gamma(v) \subseteq_{\sigma(v)} \alpha(v)$. Initially the unwinding contains only the root v_0, labeled with the concrete trace $\mathit{InitialAbstraction}(\mathcal{S})$. Concrete label $\gamma(v)$ is obtained as a result of a chain of applications of causal transitions on the path from v_0 to v. Abstract label $\alpha(v)$, on the other hand, represents conditions, sufficient to repeat all unwinding steps in the subtree, originating at v; it is obtained by propagating up the premises of causal transitions, applied at the subtree vertices.

At each iteration of the algorithm main loop we select some vertex v from the queue Q of unexplored tableau leaves. First, we try to cover v by some

Algorithm 1. Causality-based Verification

Input : Transition system $\mathcal{S} = \langle \mathcal{V}, T, init, error \rangle$
Output: **safe**/**unsafe**
Data: Trace tableau $\langle \Upsilon, \rightsquigarrow, \alpha, \mu, \sigma \rangle$, where $\Upsilon = \langle V, F, \gamma, \delta, \lambda \rangle$, queue $Q \subseteq V_L$, premise of last causal transition $\tau_\triangleleft \in \mathbb{A}$, trace morphism $\xi : \tau_\triangleleft \to \mathbb{A}$

begin
- set $V \longleftarrow \{v_0\}$, $\gamma(v_0) \longleftarrow$ *InitialAbstraction*$(\mathcal{S})$
- set $Q \longleftarrow \{v_0\}$, all of $\{F, \rightsquigarrow, \alpha, \mu, \sigma, \delta, \lambda\} \longleftarrow \emptyset$
- **while** Q *not empty* **do**
  - take some v from Q
  - **if** $\exists\, v' \in V_N,\ \sigma' : \alpha(v') \to \gamma(v)\,.\,\gamma(v) \subseteq_{\sigma'} \alpha(v')$ *is forgetful* **then**
    - add (v, v') to $\rightsquigarrow$
    - set $\delta(v)_\triangleleft \longleftarrow \alpha(v')$, $\tau_\triangleleft \longleftarrow \alpha(v')$, $\xi \longleftarrow \sigma'$
  - **else**
    - set $L \longleftarrow$ *Linearize*$(\gamma(v))$
    - **if** *Concretizable*(L) **then**
      - **return** ***unsafe***
    - **else**
      - set $\langle \tau_\triangleleft, \xi \rangle \longleftarrow$ *Refine*(v, L)
  - put children of v into Q
  - *PropagateUp*$(v, \tau_\triangleleft, \xi)$
- **return** ***safe***

In: vertex v, linear trace $L = \langle N, E, \nu, \eta \rangle$
Out: $\langle$premise $\tau_\triangleleft$, trace morphism $\xi\rangle$
begin
- $\langle N' \subseteq N, E' \subseteq E \rangle \longleftarrow$ *ExtractConflict*(L)
- **if** $\exists\, o_1, o_2 \in N' \cup E'\,.\, o_1 \| o_2$ **then**
  - *OrderSplit*(o_1, o_2)
- **else**
  - **switch** $|N'|$ **do**
    - **case** *1*
      - *Contradiction*(v, n_1)
    - **case** *2*
      - $\phi =$ *Interpolate*$(\eta(n_1)\,;\, \eta(n_2)')$
      - *NecessaryAction*(v, n_1, n_2, ϕ)
    - **otherwise**
      - $\phi =$ *Interpolate*$(\eta(n_1) \wedge \ldots \wedge \eta(n_{k-1})^{k-1}\,;\, \eta(n_k)^k)$
      - *ActionSplit*(v, n_{k-1}, ϕ)
- **return** $\langle$*premise* $\tau_\triangleleft$ *of used causal transition, trace morphism* $\xi : \tau_\triangleleft \to L\rangle$

Function Refine

In: vertex v, premise $\tau_\triangleleft$, trace morphism $\xi : \tau_\triangleleft \to \gamma(v)$
begin
- **if** $\nexists\, \chi = \langle \chi_N, \chi_E \rangle : \tau_\triangleleft \to \alpha(v)\,.\, \xi = \sigma \circ \chi$ **then**
  - **foreach** $o \in \gamma(v)\,.\, \exists\, o' \in \tau_\triangleleft\,.\, o = \xi(o') \wedge \nexists\, o'' \in \alpha(v)\,.\, o = \sigma(o'')$ **do**
    - add o' to $\alpha(v)$, and (o', o) to $\sigma(v)$
- let $\chi = \langle \chi_N, \chi_E \rangle : \tau_\triangleleft \to \alpha(v)\,.\, \xi = \sigma \circ \chi$
- **if** $\alpha(v) \not\subseteq_\chi \tau_\triangleleft$ **then**
  - **foreach** $n\,.\, \eta(\chi_N(n)) \not\Longrightarrow \eta(n)$ **do**
    - set $\eta(\chi_N(n)) \longleftarrow \eta(\chi_N(n)) \wedge \eta(n)$
  - **foreach** $e\,.\, \nu(\chi_E(e)) \not\Longrightarrow \nu(e)$ **do**
    - set $\nu(\chi_E(e)) \longleftarrow \nu(\chi_E(e)) \wedge \nu(e)$
- **foreach** $(v_c, v) \in \rightsquigarrow$ **do**
  - **if** $\alpha(v_c) \subseteq_{\mu(v_c)} \alpha(v)$ *not forgetful* **then** remove (v_c, v) from $\rightsquigarrow$ put v_c into Q
- **if** $\exists$ *parent* $v'\,.\, (v', v) \in F$ **then**
  - let $\delta' : \gamma(v') \to \gamma(v)$
  - set $\langle \tau_\triangleleft, \xi \rangle \longleftarrow$ *Pullback*(δ', ξ)
  - *PropagateUp*$(v', \tau_\triangleleft, \xi)$

Procedure PropagateUp

other vertex v': this can be done if the concrete trace of v is included in the abstract trace of v' (thus, all causal transitions at v' subtree can be repeated), and, moreover, the inclusion is forgetful, i.e. it "forgets" some action on the left or on the right from the trace.

If the covering attempt was unsuccessful, we linearize the concrete trace of v. If the linear trace L is concretizable - we have found a concrete error trace, and the algorithm terminates with **unsafe**; otherwise there is a conflict in L, and we proceed to the refinement phase, where some causal transition is applied to v.

At the end of each iteration we put into queue Q all children of v, added during the refinement phase, and propagate the premise of the applied causal transition up the unwinding tree. We finish the main loop of the algorithm as soon as queue Q becomes empty: in that case the only uncovered leaf vertices left are contradictory, and the tableau is complete; the algorithm returns **safe**.

The refinement function *Refine* is the main point of application of a wide spectrum of optimizations and specializations possible for causalty-based verification. Here we show one possible instantiation for *Refine*, which operates on a non-linearizable concurrent trace. First, it extracts a conflict from the trace (for example, by computing an unsatisfiable core): a minimal subtrace, which is still non-linearizable. If the subtrace contains some unordered actions or edges, an *order split* is applied, which forces a particular order. Otherwise we consider different cases with respect to the number of actions in the non-linearizable subtrace. If there is only one action, the trace is surely contradictory. If there are two actions, we apply the *necessary action*, which tries to repair the conflict by introducing new action in the middle. For that purpose we compute the Craig interpolant between contradictory actions. Finally, if there are more than two actions in the subtrace we shorten the subtrace by splitting the last but one action with the Craig interpolant between the last action and the rest of them. Each of the causal transitions applied returns its premise and the mapping of it into the concrete trace of the vertex: they are used later for propagation.

The propagation of premises is done in procedure *PropagateUp*, and explained graphically in the left part of Figure 4. Given as input the premise $\tau_\triangleleft$ and the mapping ξ of it to the concrete label $\gamma(v)$, the procedure adds missing components to the abstract label $\alpha(v)$. This is done in two stages: first, the objects (actions or edges), which are present in $\tau_\triangleleft$, but missing in $\alpha(v)$ are inserted into $\alpha(v)$, producing such $\alpha(v)'$ that there is a mapping $\chi : \tau_\triangleleft \to \alpha(v)'$; second, their labels in $\alpha(v)'$ are adjusted in such a way, that $\gamma(v) \subseteq_\chi \alpha(v)''$ holds. Because the abstract label $\alpha(v)$ becomes more concrete, previous covering by that vertex may stop to hold; they are checked and uncovered as needed. Finally, the premise is propagated up in the tableau to vertex v', by constructing the premise $\tau'_\triangleleft$ and the mapping $\xi' : \tau'_\triangleleft \to \gamma(v')$ as a pullback object and arrow of two arrows ξ and δ', where δ' is a direct transformation of $\gamma(v')$ to $\gamma(v)$. Then *PropagateUp* is called recursively with this new premise and mapping; the propagation process terminates either when the root vertex is reached, or when the abstract label $\alpha(v)$ already contains all objects used in the premise, i.e. when the inclusion $\alpha(v) \subseteq \tau_\triangleleft$ holds.

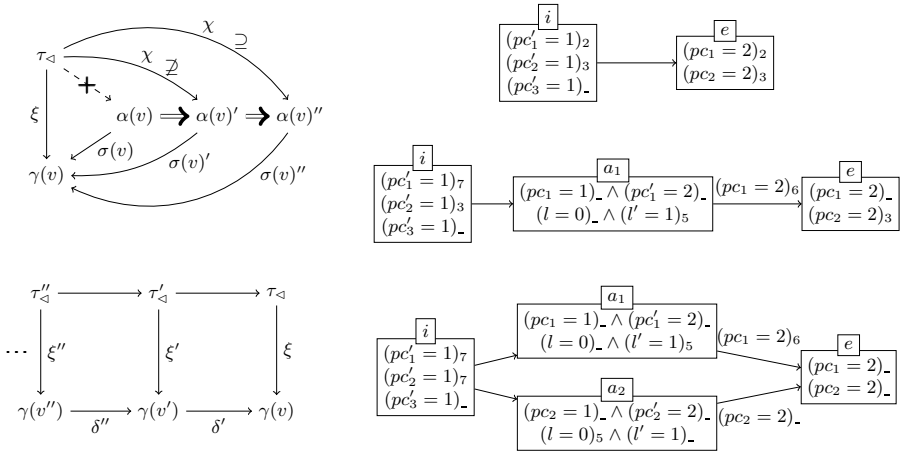


Fig. 4. Upward propagation of premises in trace tableau. *Top left*: calculation of abstract label $\alpha(v)$ in procedure *PropagateUp*. *Bottom left*: pullback construction, propagation of premise $\tau_\triangleleft$ to parent vertices. *Right*: abstract labels for the first three vertices of the tableau from motivating example.

The right part of Figure 4 depicts the abstract labels of the first three vertices of the tableau, obtained after the execution of seven steps from the motivating example. Each conjunct of action or edge labels is marked with the number, showing at which step of the execution this conjunct was included into the abstract label; the undescore sign means that this conjunct is not present in the abstract label after all seven steps. For example, for the top right trace in Figure 4, the conjuncts $(pc'_1 = 1)$ and $(pc_1 = 2)$ were included in the abstract label in the second step, the conjuncts $(pc'_2 = 1)$ and $(pc_2 = 2)$ in the third step, and the conjunct $(pc'_3 = 1)$ was never used for all seven execution steps.

6 Polynomial-Time Verification for Programs with Locks

To demonstrate the advantages of causality-based verification, let us return again to the class of multi-threaded programs with locks, which we considered as a motivating example in Section 2. Standard model checking approaches require exponential, with respect to the number of threads, time and space to prove safety of such programs. In [7], a counterexample-guided refinement algorithm based on cartesian abstraction with exception sets is developed, which is capable to solve in polynomial time the safety problem for the restricted class of programs with locks. The restricted class allows only one lock variable, prohibits nesting/intersection of critical sections, and disallows control flow transfers. Alexander Malkis posed the following:

> **Open Problem ([7], p.65).** Is the most general locks class polynomially verifiable for a fixed number of locks?

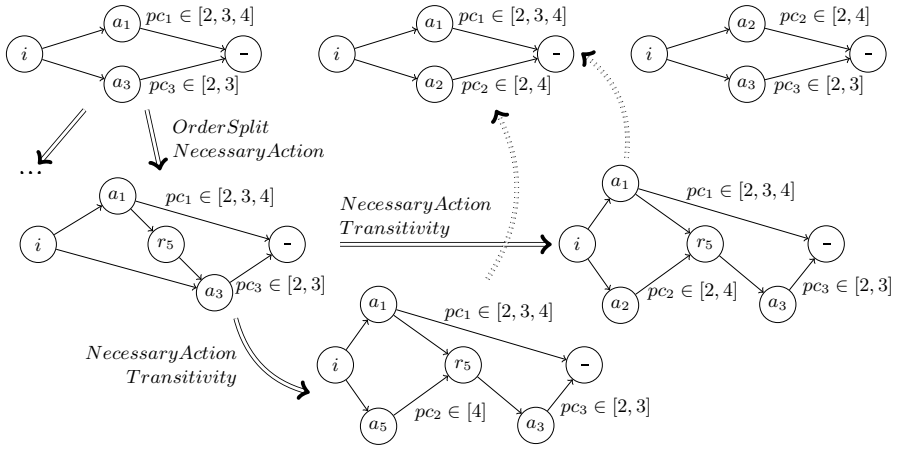


Fig. 5. Part of the causal trace tableau for the example multi-threaded program with locks

Here we settle this question affirmatively; moreover, our trace refinement algorithm finds safety proofs for the most general class of programs with locks using only polynomial time and space with respect both to the number of threads and to the number of locks.

Our algorithm starts its computation as shown in Section 2. After several initial steps, the causal tableau starts looping in the repetitions of the same concurrent scenario, shown in Figure 5: two threads enter (actions a_1, a_3) and stay (restrictions $pc_1 \in [2,3,4]$ and $pc_3 \in [2,3]$) in their critical sections for the same lock variable. Because all acquire actions satisfy the constrain $l = 0 \wedge l' = 1$, any ordering of them produces the conflict $l' = 1 \not\downarrow l = 0$. The algorithm applies the *necessary action* causal transition, and inserts a release action, which can be instantiated to transitions r_1, r_2, r_3, and r_5; in the example we consider only transition r_5. Now the trace contains the conflict $pc'_2 = 1 \not\downarrow pc_2 = 4$ between actions i and r_5. There are two possible paths between locations 2 and 4 of the second trace; by inserting necessary actions on both alternative paths, we finally introduce actions a_2 and a_5 respectively. Both are acquire actions and we again repeat the concurrent scenario with two threads trying to enter critical sections; thus, the new tableau vertices are covered. Moreover, there is an action (a_3 in this case), which the covering forgets; thus, the covering is forgetful.

It is easy to check, that the number of vertices in the tableau is proportional to the cubic power of the number of critical sections, while the size of the concurrent traces, labeling the vertices, is independent of the number of threads, critical sections, and locks. The execution of our algorithm takes at most quadratic time with respect to the number of vertices; thus, we have the following:

Theorem 4. *Causality-based verification algorithm proves the safety of the most general class of multi-threaded programs with binary locks in deterministic polynomial time and space with respect to the number of threads and locks.*

7 Conclusion

We have presented a new verification procedure for concurrent systems, which analyzes causal chains in the system behavior. In our procedure, we capture the causal dependencies as Mazurkiewicz-style concurrent traces, and explore the unwinding tree of the causally related traces. Our procedure terminates as soon as all the paths in the unwinding tree are either contradictory, or are covered by other tree vertices, where the same concurrent situation was already examined.

The key ingredient that distinguishes our approach from techniques based on state space exploration or Petri net unfoldings, is that we do not restrict ourselves to only forward or backward analysis of all the transitions available at the current analysis stage. Instead, we try to build a minimal concurrent error trace, which contains only the necessary transitions on the way from initial to error states. We have demonstrated that in some cases, such as multi-threaded programs with locks, our approach reduces the verification complexity from exponential to polynomial.

The full version of the present paper with all proofs is available in [6].

Acknowledgements. The authors thank the anonymous reviewers for their valuable comments and suggestions.

References

1. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (2001)
2. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic approaches to graph transformation - part i: Basic concepts and double pushout approach. In: Rozenberg [11], pp. 163–246
3. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic approaches to graph transformation - part ii: Single pushout approach and comparison with double pushout approach. In: Rozenberg [11], pp. 247–312
4. Esparza, J., Heljanko, K.: Unfoldings - A Partial-Order Approach to Model Checking. EATCS Monographs in Theoretical Computer Science. Springer (2008)
5. Godefroid, P. (ed.): Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. LNCS, vol. 1032. Springer, Heidelberg (1996)
6. Kupriyanov, A., Finkbeiner, B.: Causality-based verification of multi-threaded programs. Reports of SFB/TR 14 AVACS 92, SFB/TR 14 AVACS (2013) ISSN: 1860-9821, `http://www.avacs.org`
7. Malkis, A.: Cartesian Abstraction and Verification of Multithreaded Programs. PhD thesis, Albert-Ludwigs-Universität Freiburg im Breisgau (2010)
8. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, New York (1995)
9. Mazurkiewicz, A.: Concurrent program schemes and their interpretations. Technical Report DAIMI PB 78, Aarhus University (1977)
10. Reisig, W.: Petri Nets – An Introduction. Springer (1985)
11. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations. Foundations, vol. 1. World Scientific (1997)

From Model Checking to Model Measuring*

Thomas A. Henzinger and Jan Otop

IST Austria

Abstract. We define the *model-measuring problem*: given a model M and specification φ, what is the maximal distance ρ such that all models M' within distance ρ from M satisfy (or violate) φ. The model measuring problem presupposes a distance function on models. We concentrate on *automatic distance functions*, which are defined by weighted automata. The model-measuring problem subsumes several generalizations of the classical model-checking problem, in particular, quantitative model-checking problems that measure the degree of satisfaction of a specification, and robustness problems that measure how much a model can be perturbed without violating the specification. We show that for automatic distance functions, and ω-regular linear-time and branching-time specifications, the model-measuring problem can be solved. We use automata-theoretic model-checking methods for model measuring, replacing the emptiness question for standard word and tree automata by the *optimal-weight question* for the weighted versions of these automata. We consider weighted automata that accumulate weights by maximizing, summing, discounting, and limit averaging. We give several examples of using the model-measuring problem to compute various notions of robustness and quantitative satisfaction for temporal specifications.

1 Introduction

Model-checking techniques have proved to be very useful in automatic verification. Typically, the verified system is modeled as a transition system, the desired properties are specified by a formula in a temporal language (Linear Temporal Logic [LTL], Computation Tree Logic[CTL]) or an ω-automaton, and a model-checking algorithm decides whether the model is correct with respect to the specification. However, knowing whether the model is correct or not, is often insufficient.

Consider the TCP handshake protocol, which is used to establish a connection between a client and a server. First, the client sends a SYN packet to the server, which replies with a SYN-ACK packet. Then, the client responds with an ACK packet. A TCP connection is established, provided that the protocol terminated.

Termination of the protocol can be verified by the standard model-checking techniques, when the communication channel is assumed to be reliable, that is,

* This work was supported in part by the Austrian Science Fund NFN RiSE (Rigorous Systems Engineering) and by the ERC Advanced Grant QUAREM (Quantitative Reactive Modeling).

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 273–287, 2013.

every sent packet is delivered in the next step. Certain faults, such as "the first server response SYN-ACK gets lost" can be encoded in the model. But, this raises doubts whether the model includes all communication faults. Another approach would be to use fairness assumptions, for example "if infinitely many packets are sent, infinitely many packets will be delivered". But, such assumptions may be too weak to guarantee termination of the protocol. We propose a more refined, quantitative approach.

We assume that any packet may get lost, but we ask quantitative questions: What is the maximal number of lost packets tolerated by the protocol? What is the maximal ratio of lost packets that guarantees liveness of the system? Such questions are instances of the model-measuring problem.

The *model-measuring problem* asks, given a model M and specification φ, what is the maximal distance ρ such that all models M' within that distance from M satisfy (or violate) φ. That distance ρ is called the *stability radius*. Figure 1 presents a geometric interpretation of the stability radius in two cases, a model M that satisfies the specification and a model N that violates it.

To determine the stability radius, it suffices to have a unary function that, for a given transition system M', specifies its distance from M. Such a function, called a *similarity measure*, is a sole input to the model-measuring problem. As inputs are required to be finite, we are interested in *automatic* similarity measures that are represented by weighted automata.

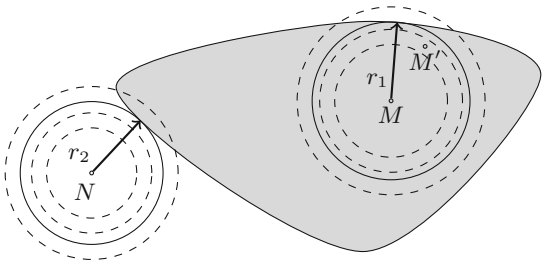


Fig. 1. A geometric interpretation of stability radii r_1 of a model M and r_2 of a model N. The shaded area represents the family of models satisfying the specification.

In the TCP handshake protocol example, a model N encodes all executions of the protocol over a reliable channel. Next, we define a similarity measure d_N so that $d_N(M) = k$ if M encodes the TCP handshake protocol that loses (up to) k packets during its execution. Then, for the specification "the protocol terminates", the model-measuring problem answers the question, what is the maximal number of lost packets that guarantees termination of the protocol?

We represent similarity measures by weighted automata; the representation depends on the type of the specification. For example, in the branching-time case, every transition system (model) M' admits the unique unrolling to a tree $t_{M'}$. Then, a weighted automaton $\mathcal{A}_{dist}$ represents a similarity measure d_M, if for every transition system M', $d_M(M')$ equals to $\mathcal{A}_{dist}(t_{M'})$, the weight of $t_{M'}$ assigned by $\mathcal{A}_{dist}$.

Similarity measures represented by weighted automata are invariant with respect to bisimilarity. This design choice is not accidental as we think that two systems should be considered similar when their outputs are similar rather than when the internal structures are similar. After all, we would consider two different implementations of the same algorithm as similar rather than two similar programs that implement different algorithms.

Having an automatic representation of d_M, we can solve the model-measuring problem. Returning to the branching-time case, a (qualitative) automata-theoretic CTL model-checking procedure works as follows. It translates $\neg\varphi$ and M to ω-tree automata $\mathcal{A}_{\neg\varphi}, \mathcal{A}_M$, where $\mathcal{A}_{\neg\varphi}$ recognizes the set of all trees that satisfy $\neg\varphi$ and $\mathcal{A}_M$ accepts only a single tree, the unrolling of M. Then, it asks for emptiness of $\mathcal{L}(\mathcal{A}_{\neg\varphi} \times \mathcal{A}_M) = \mathcal{L}(\mathcal{A}_{\neg\varphi}) \cap \mathcal{L}(\mathcal{A}_M)$. In our approach, we replace $\mathcal{A}_M$ by a weighted ω-tree automaton $\mathcal{A}_{dist}$ representing d_M, and generalize the emptiness question to its weighted counterpart, the *optimal-weight question*. That question asks for the infimum over weights of all ω-trees (ω-words) accepted by a weighted ω-tree automaton. Now, let ρ be the answer to the optimal weight question for $\mathcal{A}_{\neg\varphi} \times \mathcal{A}_{dist}$. It follows that for every $\rho' > \rho$, there is a tree accepted by $\mathcal{A}_{\neg\varphi}$ of weight at most ρ', and every M', whose distance from M is less than ρ, satisfies φ. Thus, ρ is the stability radius of φ in M. Virtually the same argument can be repeated in the linear-time case using ω-automata and ω-words.

The contribution of this paper is two-fold. First, we define the model-measuring framework (Section 3) and show that several problems studied in the literature are special cases of the model-measuring problem. Second, we give a systematic approach to modeling similarity measures using weighted automata, and corresponding algorithms based on the optimal-weight question for computing them.

The paper is organized as follows. In Section 2 we recall the standard notions of weighted and unweighted automata, define the optimal weight question and discuss its complexity in various cases. In Section 3 we define the stability radius of a model w.r.t. a specification and the model-measuring problem. We start with general definitions of these notions, which are then specialized to the ω-regular linear-time and branching-time settings, based on weighted automata. Finally, in Section 4 we discuss in depth the modeling of similarity measures and give several examples in each case.

Related work. In recent years, much attention has been given to quantitative[1] generalizations of the Boolean notion of correctness and the corresponding quantitative verification questions [2,3,14,15,18]. Here we attempt to define a unifying automata-theoretic framework to capture and compute various ways of measuring model quantities. In particular, we have succeed in subsuming the following approaches.

The robust satisfaction of an open system has been studied in [14,18]. An open system M *robustly satisfies* a CTL specification φ (according to [14]) if

[1] Note that we use the attribute "quantitative" in a non-probabilistic sense. We therefore restrict ourselves to list only non-probabilistic references.

and only if for every environment, given as an open system M', the composition $M \parallel M'$ satisfies φ (refer to [14] for the formal definition of composition). The model-measuring problem subsumes this notion of robustness (cf. Section 4).

The model-measuring problem can express *mutations on circuits* [17]. Indeed, all mutations considered in [17] just modify transition relations of automata, therefore they can be expressed by our *hypervisor* approach (cf. Example 24). In consequence, the model-measuring problem subsumes vacuity [19], coverage [10], and certain cases of fault tolerance [11].

Another approach to robustness of discrete systems has been presented in [3], where the *robustness distance* has been defined. This robustness distance can be expressed in our framework as well (cf. Example 25).

2 Preliminaries

A tree (ω-tree) t over Γ labeled by Σ is a pair (τ, L), where τ is a finite (infinite for ω-trees) prefix-closed subset of Γ^* and $L : \tau \mapsto \Sigma$ is a labeling function. For $\sigma \in \tau$, every extension $\sigma \cdot g$ of σ, where $g \in \Gamma$ and $\sigma \cdot g \in \tau$, is a *successor* of σ in (τ, L). We write $\sigma \in t$ and $t(\sigma)$ instead of $\sigma \in \tau$ and $L(\sigma)$. We usually omit Γ.

A *labeled transition system* is a quadruple $\langle S, \Sigma, E, s_0\rangle$, where S is a (finite or infinite) set of states, Σ is an alphabet, E is a relation on $S \times \Sigma \times S$ and s_0 is an initial state. All models considered in this paper are (finite or infinite) transition systems. A word (or ω-word) $w = a_1 a_2 \dots$ is a *trace* of a labeled transition system M if there is an (unlabeled) path $s_0 s_1 \dots$ in M such that for every $i \in [1, |w|]$, $(s_{i-1}, a_i, s_i) \in E$. We say that an ($\omega$-)tree (τ, L) (over $S \times (\Sigma \cup \{\epsilon\})$) labeled by $\Sigma \cup \{\epsilon\}$ is the *unrolling* of a transition system $M = \langle S, \Sigma, E, s_0\rangle$ if τ is the union of all finite labeled paths $\langle s_0, \epsilon\rangle\langle s_1, a_1\rangle \dots \langle s_k, a_k\rangle$ through M such that for every $i \in [1, k]$, $(s_{i-1}, a_i, s_i) \in E$, and $L(\langle s_0, \epsilon\rangle \dots \langle s_k, a_k\rangle) = a_k$.

2.1 Automata

A *(nondeterministic) automaton* is a tuple $(\Sigma, Q, Q_0, \delta, F)$, where Σ is an alphabet, Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation and F is an acceptance condition (finite, Büchi, ...).

A *run* π of an automaton $\mathcal{A}$ on $w = a_1 a_2 \dots$ is a sequence of states such that $\pi(0) \in Q_0$ and for every $i \in [1, |w|]$, $(q_{i-1}, a_i, q_i) \in \delta$. A run π is *accepting* if it satisfies the acceptance condition F, e.g., in the Büchi case: there is $q \in F$ that occurs infinitely often in π.

A *(nondeterministic) (ω-)tree automaton with varying degree* (bounded by N) [20] is a tuple $(\Sigma, Q, Q_0, \delta, F)$, where Σ is an alphabet, Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, $\delta \subseteq \bigcup_{k=1}^{N} (Q \times \Sigma) \times Q^k$ is a transition relation and F is an acceptance condition (finite, Büchi, parity, ...).

A *run* π of an automaton $\mathcal{A}$ on an (ω-)tree $t = (\tau, L)$ is an (ω-)tree (τ, L') labeled by Q such that $\pi(\epsilon) \in Q_0$ and for every $\sigma \in t$, if $deg(\sigma) = k$ and $\sigma \cdot g_1$, $\dots$, $\sigma \cdot g_k$ are all successors of σ in t, then $(\pi(\sigma), t(\sigma), \langle \pi(\sigma \cdot g_1), \dots, \pi(\sigma \cdot g_k)\rangle) \in \delta$. A run π is *accepting* if it satisfies the acceptance condition F, e.g., in the Büchi case: along every infinite path there is a state from F that occurs infinitely often.

A weighted (Büchi, parity, Büchi-tree, ...) automaton is an automaton whose transitions are labeled by natural numbers called weights. Formally, a weighted automaton $\mathcal{A}$ is a tuple $(\Sigma, Q, Q_0, \delta, F, C)$ such that $(\Sigma, Q, Q_0, \delta, F)$ is an automaton and $C : \delta \mapsto \mathbb{N}$.

A *weighting scheme* is a function that maps runs to real numbers, called *weights*. The weight of an ω-word w (ω-tree t) assigned by the automaton $\mathcal{A}$ according to a weighting scheme f, denoted by $L^f_{\mathcal{A}}(w)$, is the infimum of the set of weights of all accepting runs of $\mathcal{A}$ on the ω-word w (the ω-tree t) weighted by f. ω-words (ω-trees) that are rejected by $\mathcal{A}$ have infinite weight. Often, a particular weighting scheme is irrelevant in reasoning, as long as it is fixed through a proof; in such cases we shall omit it.

The emptiness question for non-weighted automata extends to the following question in the weighted case:

Definition 1. *Let f be a weighting scheme. The* optimal-weight question *for f asks, given a weighted automaton $\mathcal{A}$, to compute the infimum of $L^f_{\mathcal{A}}(w)$ over all ω-words (ω-trees).*

Remark 2. The dual to the optimal-weight question is to find the supremum of $L^f_{\mathcal{A}}(w)$ over all ω-words w. Its decision versions have been referred to as the limitedness problem [21] or the universality problem for weighted automata [7]. They are usually much harder than the optimal-weight problem (see [5] for undecidability results).

2.2 Weighting Schemes for ω-words

Let $\mathcal{A}$ be a weighted automaton and π be its run. Denote by $wt(\pi, i)$ the weight of the ith transition in π. We consider the following weighting schemes:

1. $\textsc{Sum}(\pi) = \sum_{i=1}^{\infty} wt(\pi, i)$, the sum,
2. $\textsc{Max}(\pi) = \max_{i=1}^{\infty} wt(\pi, i)$, the maximum,
3. $\textsc{Disc}_\lambda(\pi) = (1 - \lambda) \sum_{i=1}^{\infty} \lambda^i wt(\pi, i)$, the discounted sum, where $\lambda \in (0, 1)$ is a *discount factor*,
4. $\textsc{LimAvg}(\pi) = \liminf_{k \to \infty} \frac{1}{k} \sum_{i=1}^{k} wt(\pi, i)$, the limit average.

These weighting schemes admit efficient algorithms computing the optimal-weight question:

Theorem 3. *([13,22]) Let f be one of* $\textsc{Sum}, \textsc{Max}, \textsc{Disc}_\lambda, \textsc{LimAvg}$. *The optimal-weight question for f and a weighted Büchi automaton $\mathcal{A}$ can be computed in polynomial time in $|\mathcal{A}|$.*

2.3 Weighting Schemes for ω-trees

In the ω-tree case, we consider two families of weighting schemes, Sup and Acc. The Sup weighing schemes are derived from ω-words weighting schemes; every path in a run on an ω-tree is weighted according to an ω-words weighting scheme,

and the weight of the run is supremum over weights of its all paths. We consider the following SUP weighting schemes: SUPSUM, SUPMAX, SUPDISC$_\lambda$ and SUPLIMAVG.

The ACC family is obtained by accumulating weights over all paths. Given a run π over an ω-tree t and $\sigma \in t$, we define: ***(i)*** $wt(\pi, \sigma)$ as the weight of a transition at σ in the run π, ***(ii)*** the contribution of σ in π, denoted by $\mu(\pi, \sigma)$, as follows: $\mu(\pi, \epsilon) = 1$, and for every successor $\sigma \cdot g$ of σ, $\mu(\pi, \sigma \cdot g) = \frac{1}{deg(\sigma)}\mu(\pi, \sigma)$.

We define the following weighting schemes:

1. $\text{ACCSUM}(\pi) = \sum_{\sigma \in \pi} \mu(\pi, \sigma) wt(\pi, \sigma)$, the accumulated sum,
2. $\text{ACCDISC}_\lambda(\pi) = (1 - \lambda) \sum_{\sigma \in \pi} \lambda^{|\sigma|} \mu(\pi, \sigma) wt(\pi, \sigma)$, the accumulated discounted sum, where $\lambda \in (0, 1)$ is a *discount factor*,
3. $\text{ACCLIMAVG}(\pi) = \liminf_{k \to \infty} \frac{1}{k} \sum_{\sigma \in \pi, |\sigma| \leq k} \mu(\pi, \sigma) wt(\pi, \sigma)$, the accumulated limit average.

Theorem 4. *([1,4,6,8,9,23]) Let f be one of* SUPSUM, SUPMAX, SUPDISC$_\lambda$, SUPLIMAVG, ACCSUM, ACCDISC$_\lambda$ *or* ACCLIMAVG. *The optimal-weight question for f and a weighted Büchi-tree automaton $\mathcal{A}$ can be computed in polynomial time in $|\mathcal{A}|$.*

Table 1. The complexity of the optimal-weight question for weighted Büchi and Büchi-tree automata. $(*)$ indicates that the algorithm work in polynomial time under assumption that the weights are given in unary notation.

	ω-words	ω-trees	
		SUP	ACC
SUM	$O(n \log n)$	PTIME	PTIME
MAX	$O(n \log n)$	PTIME	—
DISC$_\lambda$	PTIME	PTIME$^{(*)}$	PTIME
LIMAVG	PTIME	PTIME$^{(*)}$	PTIME

Remark 5. The optimal-weight question can be solved for parity ω-word and ω-tree automata with all weighting schemes from Theorems 3 and 4. However, its complexity in the parity case increases from PTIME to the complexity of solving parity games.

2.4 Automatic (Weighted) Relations

The *convolution* of ω-words w_1, w_2, denoted by $w_1 \otimes w_2$, is an ω-word over $\Sigma \times \Sigma$ such that the ith letter of $w_1 \otimes w_2$ is a pair of the ith letters of w_1, w_2.

A *weighted relation* is a generalization of the usual relation by allowing the characteristic function to range over $\mathbb{R}^+ \cup \{\infty\}$. A (binary) weighted relation S is an *automatic weighted relation* if there is a weighted automaton $\mathcal{A}_S$ that *computes* S, i.e., for all $w_1, w_2 \in \Sigma^*$, $w_1 S w_2 = \mathcal{A}_S(w_1 \otimes w_2)$.

The notion of automatic weighted relations straightforwardly extends on ω-trees.

3 The Model-Measuring Framework

Correctness of a system w.r.t. a specification is, like membership, a qualitative property; the system is correct or not. However, membership of a point p in a region R has a natural quantitative extension called the *stability radius*. It is defined as the distance between p and the border of R (cf. Fig. 1). It has been widely used in the decision-making community [16]. Assuming that we are given a distance function d defined on transition systems, we adapt the stability radius to the model-checking setting. Basically, we ask for stability radius of a transition system in the region of all transition systems satisfying a specification.

The definitions in this section are independent of a particular logic. They refer to a *specification* which is not yet instantiated. It will be instantiated in Sections 3.1 and 3.2.

Definition 6. *Let d be a distance defined on transition systems. For a transition system M and a specification P, the* stability radius *of P in M (w.r.t. the distance d), denoted by $sr_d(M, P)$, is defined as follows:*

(i) if $M \models P$, $sr_d(M,P) = \sup\{\rho \geq 0 : \forall M'(d(M,M') < \rho \Rightarrow M' \models P)\}$,
(ii) if $M \models \neg P$, $sr_d(M,P) = sr_d(M, \neg P)$,
(iii) otherwise, $sr_d(M,P) = 0$.

In order to determine the stability radius of P in M we only need to know distances between a (fixed) M and other transition systems; it suffices to have a unary function d_M, defined as $d_M(M') = d(M,M')$, which encodes essential information about M and d. We call such a function d_M a *similarity measure*. Observe that any function satisfying $d_M(M) = 0$ and $d_M(M') \geq 0$ is a valid similarity measure as we can find a distance d defining it. However, we are interested only in similarity measures that are semantically defined, i.e., those that depend only on the behavior of the transition system (the set of traces), not on its structure.

We define the stability radius of P in M w.r.t. a similarity measure d_M, $sr_{d_M}(M,P)$, as the stability radius w.r.t. any distance compatible with d_M.

We define the *model-measure* on the basis of the stability radius by scaling the value the stability radius from $[0,\infty]$ to $[\frac{1}{2},1]$ if $M \models P$, and $[0,\frac{1}{2}]$ otherwise.

Definition 7. *The* model-measuring problem *is defined as follows: given a similarity measure d_M and a specification P, compute $[P]_{d_M}$ defined as follows:*

(i) if $M \models P$, $[P]_{d_M} = 1 - 2^{-sr_{d_M}(M,P)-1}$ $(\in [\frac{1}{2},1])$,
(ii) if $M \models \neg P$, $[P]_{d_M} = 1 - [\neg P]_{d_M}$ $(\in [0,\frac{1}{2}])$,
(iii) otherwise, $[P]_{d_M} = \frac{1}{2}$.

Consider a specification given by a temporal (LTL or CTL) formula φ. The model-measure is compatible with conjunction and implication, i.e., $[\varphi_1 \wedge \varphi_2]_{d_M} = min([\varphi_1]_{d_M}, [\varphi_2]_{d_M})$ and $\varphi_1 \Rightarrow \varphi_2$ implies $[\varphi_1]_{d_M} \leq [\varphi_2]_{d_M}$. Observe that for every similarity measure d_M, $[\varphi]_{d_M} = 1$ if φ is a tautology, as $sr_{d_M}(M,\varphi) = \infty$ and $1 - 2^{-\infty-1} = 1$, and $[\varphi]_{d_M} = 0$ if φ is inconsistent. Values of formulae that are neither tautologies nor inconsistent depend on the choice of a similarity measure.

Example 8. Consider a transition system M modeling two parties communicating through a channel, where every sent packet is delivered in the next state. We define a similarity measure d_M, such that $d_M(M') = k$ if M' models two parties that follow the same protocol as in M, but up to k packets sent through the channel get lost. We shall return to this example in Section 4.

In the following, we discuss specialization of the model-measuring problem for ω-regular linear-time and branching-time specifications.

3.1 Model Measuring ω-regular Linear-Time Specifications

An *ω-regular linear-time specification* P is a subset of Σ^ω, the set of all correct traces. We assume that P is given by a Büchi automaton $\mathcal{A}_P$ recognizing its complement, i.e., an ω-word w violates P iff $\mathcal{A}_P$ accepts w. E.g. a Linear Temporal Logic (LTL) formula φ can be translated to a Büchi automaton $\mathcal{A}_{\neg\varphi}$ recognizing ω-words that satisfy $\neg\varphi$. The automaton $\mathcal{A}_P$ can be regarded as a weighted automaton with all weights 0. Next, we say that a transition system M satisfies a linear-time specification P if all its traces satisfy P, or equivalently, the language of all traces of M and $\mathcal{L}(\mathcal{A}_P)$ are disjoint.

We proceed alike with similarity measures. We define similarity measures on ω-words, then we extend the definition to transition systems.

Definition 9. *A (linear-time) similarity measure d_M is* automatic *iff there is a weighted automaton $\mathcal{A}_{dist}$ and a weighting scheme $f \in \{\textsc{Sum}, \textsc{Max}, \textsc{Disc}_\lambda, \textsc{LimAvg}\}$ such that for every transition system M', $d_M(M') = \sup\{\mathcal{A}^f_{dist}(w) : w$ is a trace of $M'\}$.*

Example 10. Consider a finite transition system M. Let $\mathcal{A}_{dist}$ be a weighted automaton that contains M and has a single additional state $q_\perp \notin M$. There are transitions, labeled by every letter, from every state of $\mathcal{A}_{dist}$ to $q_\perp$; each such transition has the weight 1. The state $q_\perp$ is accepting, but it has only self-loops of weight 1. All transitions of M are weighted by 0. The automaton $\mathcal{A}_{dist}$ weighted by $\textsc{Disc}_\lambda$ (with $\lambda \in (0,1)$) assigns the weight 0 to all traces of M. If w is not a trace of M, $\mathcal{A}^{\textsc{Disc}_\lambda}_{dist}(w) = (1-\lambda)\sum_{i=k}^{\infty} \lambda^i = \lambda^k$, where k is the length of the longest common prefix of w and any trace of M. Observe that for a transition system M', $d_M(M')$ is equal to λ^K, where K is the maximal number such that every trace of M' agree on the first K letters with some trace of M.

We discuss constructions of automatic similarity measures in Section 4. Now, we assume that a weighted automaton $\mathcal{A}_{dist}$ computing d_M is given and we show how to use it to compute $[P]_{d_M}$.

Consider the usual model-checking problem for LTL specifications: given a transition system M and an LTL formula φ, decide whether $M \models \varphi$. An automata-based model-checking procedure constructs two ω-automata: $\mathcal{A}_M$ accepting all traces of M, and $\mathcal{A}_{\neg\varphi}$ accepting all ω-words that violate φ. ω-words accepted by both, $\mathcal{A}_M$ and $\mathcal{A}_{\neg\varphi}$, are counterexamples to the statement $M \models \varphi$. Thus, the model-checking problem $M \models \varphi$ reduces to emptiness of

$\mathcal{L}(\mathcal{A}_M \times \mathcal{A}_{\neg\varphi}) = \mathcal{L}(\mathcal{A}_M) \cap \mathcal{L}(\mathcal{A}_{\neg\varphi})$. In order to compute the model-measure, we follow the same scheme.

Since the specification is already given by the automaton $\mathcal{A}_P$ recognizing its complement, we simply replace $\mathcal{A}_M$ with $\mathcal{A}_{dist}$ and compute the optimal-weight of the cross product $\mathcal{A}_{dist} \times \mathcal{A}_P$. This automaton is defined as the usual cross product of Büchi automata, but the weight of every transition is the weight of its first component in $\mathcal{A}_{dist}$. Observe that $\mathcal{A}_{dist} \times \mathcal{A}_P(w) = \mathcal{A}_{dist}(w)$ if $w \in \mathcal{L}(\mathcal{A}_P)$ and $\mathcal{A}_{dist} \times \mathcal{A}_P(w) = \infty$ otherwise. The optimal-weight of $\mathcal{A}_{dist} \times \mathcal{A}_P$ is precisely the value of $sr_{d_M}(M, P)$. Indeed, assume that $M \models P$ and consider, for every $\rho > 0$, an (infinite) transition systems M_ρ, such that the traces of M_ρ are all ω-words w with $\mathcal{A}_{dist}(w) \leq \rho$. Clearly, $d_M(M_\rho) = \rho$. Observe that if an ω-word w has the weight ρ assigned by $\mathcal{A}_{dist}$, i.e., $\mathcal{A}_{dist}(w) = \rho$, and $\mathcal{A}_P$ accepts w, then M_ρ violates P and $sr_{d_M}(M, P) \leq \rho$. Conversely, if $\mathcal{A}_P$ rejects all ω-words w with $\mathcal{A}_{dist}(w) < \rho$, then for every $\rho' < \rho$, $M_{\rho'} \models P$ and $sr_{d_M}(M, P) \geq \rho$. The case where $M \models \neg P$ is symmetric, and the case $M \not\models P$ and $M \not\models \neg P$ is trivial.

Theorem 11. *Assume that (linear-time) similarity measures and ω-regular linear-time specifications are given by weighted Büchi automata with one of the weighting schemes* Sum, Max, Disc$_\lambda$ *or* LimAvg. *Then, the model-measure $[P]_{d_M}$ can be computed in polynomial time in the size of both automata representing d_M and P.*

The size of $\mathcal{A}_{dist} \times \mathcal{A}_P$ is quadratic in $|\mathcal{A}_{dist}| + |\mathcal{A}_P|$. Since the optimal-weight question for Disc$_\lambda$, LimAvg weighting schemes is equivalent to computing the value of the optimal strategy in a Markov decision process, and the latter is solved by linear programming, the optimal-weight questions for Disc$_\lambda$ and LimAvg are solved in polynomial time assuming that arithmetical operations have constant costs. The question, whether linear programming, and in consequence the optimal-weight questions for Disc$_\lambda$ and LimAvg, admit polynomial-time algorithms when costs of arithmetic operations are proportional to lengths of their arguments, is still open.

3.2 Model Measuring ω-regular Branching-Time Specifications

An *ω-regular branching-time specification* P is a subset of ω-trees labeled by Σ, the set of all valid computation trees. We assume that P is given by a Büchi-tree automaton $\mathcal{A}_P$ recognizing the set of all ω-trees that violate P. The automaton $\mathcal{A}_P$ is an automaton over trees with varying (but bounded) degree. It can be regarded as a weighted automaton with all weights 0. Next, we proceed as in the linear-time case.

Surprisingly, the definition of similarity measure is simpler in the branching-time than in the linear-time case. Since every transition system M' has the unique unrolling to an ω-tree $t_{M'}$, the similarity measure of a transition system M' is defined directly as the weight of its unrolling $t_{M'}$.

Definition 12. *A (branching-time) similarity measure d_M is* automatic *iff there is a weighted ω-tree automaton $\mathcal{A}_{dist}$ and a weighting scheme f from Theorem 4 such that for every transition system M', $d_M(M') = \mathcal{A}^f_{dist}(t_{M'})$.*

Again, by virtually the same argument as in the linear-time case, the optimal weight of $\mathcal{A}_{dist} \times \mathcal{A}_P$ is equal to $sr_{d_M}(M, P)$.

Theorem 13. *Assume that (branching-time) similarity measures and ω-regular branching-time specifications are given by weighted Büchi-tree automata. Then, the model-measure $[P]_{d_M}$ can be computed in polynomial time in the size of both automata representing d_M and P.*

Remark 14. Recall that we assume that weights are given in unary notation. It is an open problem whether mean-payoff and discounted-payoff games, and in consequence the optimal-weight question for SupDisc$_\lambda$ and SupLimAvg, admit polynomial-time algorithms if weights are given in binary notation.

Remark 15. Let φ be a CTL formula. In order to compute the model-measure of φ, φ has to be translated to a non-deterministic Büchi-tree automaton $\mathcal{A}_{\neg\varphi}$ recognizing all ω-trees (of bounded degree) that violate it. Such an automaton has exponential size in $|\varphi|$. Thus, Theorem 13 yields an exponential-time algorithm computing the model-measure of a CTL formula, whereas CTL model checking has a linear-time algorithm. Unfortunately, the exponential blow-up cannot be avoided as the satisfiability problem for CTL, which is $EXPTIME$-complete, reduces to model-measuring for CTL (even with a fixed d_M). Consider a similarity measure d_M, which is finite for every transition system based on the full binary ω-tree. Observe that $[\varphi]_{d_M} < 1$ iff $\neg\varphi$ is satisfiable over the class of models based on the full binary ω-tree. The satisfiability problem for CTL, even restricted to models based on the full binary ω-tree, is $EXPTIME$-complete.

Remark 16. Theorem 13 can be generalized to parity tree automata. The optimal-weight question can be solved for parity automata over the same weighting schemes as in the Büchi case. The complexity of those algorithms is higher, but it matches the complexity of solving parity games.

3.3 Undecidable Model Measuring

We have shown that the model-measure of a linear (or branching-time) specification can be computed for automatic similarity measures. It may seem to be a narrow class of similarity measures, but even slight extensions of this class make the model-measuring problem undecidable.

Let Σ be an alphabet and let S be a relation on $\Sigma \times \Sigma$ denoting *admissible* pairs of letters. Consider a function f_M^S such that $f_M^S(w)$ is the minimal number of transpositions of admissible adjunct letters of w necessary to transform w to a trace of M. The function f_M^S is a variant of sorting, where some letters cannot be swapped. Although f_M^S cannot be computed by an automaton, as it requires unbounded memory, for every ω-word w, $f_M^S(w)$ is computable in polynomial time.

Theorem 17. *There exist M, S such that for the similarity measure defined as $d_M(M') = \sup\{f_M^S(u) : u \text{ is a trace of } M'\}$, the problem: given an LTL formula φ, decide whether $[\varphi]_{d_M} = 1$, is undecidable.*

4 Similarity Measures for ω-regular Specifications

In this section we present a systematic approach to the construction of automatic similarity measures. They will be constructed from the transition system M by relatively simple adjustments rather than modifications of M itself. The system M is usually complex, therefore modifying its internal structure is a complicated and error-prone task.

One way to construct similarity measures without modifying M itself is to employ automatic weighted relations. Let $\mathcal{A}_M$ be an automaton that recognizes the set of traces of M and let R be an automatic weighted relation computed by $\mathcal{A}_R$. A similarity measure d_M, defined by $d_M(w) = \inf\{vRw : v \text{ is a trace of } M\}$ on ω-words, and $d_M(M') = \sup\{d_M(w) : w \text{ is a trace of } M'\}$, is an automatic similarity measure. Indeed, consider a weighted automaton $\mathcal{A}_{dist}$ that, while running on the ω-word w, guesses a trace w of M on the fly and computes R by simulating $\mathcal{A}_R$. Since the weight of an ω-word is the infimum over the weights of its runs, $\mathcal{A}_{dist}(w) = \inf\{vRw : v \text{ is a trace of } M\}$ and $\mathcal{A}_{dist}$ computes d_M.

Observe that $\mathcal{A}_{dist}$ can be constructed from automata $\mathcal{A}_M$ and $\mathcal{A}_R$ in a uniform way, i.e., independently of their internal structure. This is the main advantage of that approach, but this also makes it unsuitable. To see that, suppose that an automatic weighted relation R^{E8} computes the similarity measure from Example 8. After the first packet is lost, the system (from Example 8) is in the state that is not reachable in a valid execution and a corrupt trace is not related to any valid trace of M. Thus, an automaton computing R^{E8} would have to simulate M. In consequence, it would have to remember all states of M, which is precisely what we want to avoid.

We suggest a compromise between uniformity and expressiveness. In our approach the structure of $\mathcal{A}_M$ is unaffected, but its execution is governed by an external component, called the *hypervisor*.

Definition 18. *Let $\mathcal{A}_M = (\Sigma, Q_M, Q_{0,M}, \delta_M, F_M, C_M)$ be a weighted automaton. A* hypervisor *H for $\mathcal{A}_M$ is a triple $(\mathcal{A}_H, \tau_H, \Gamma_H)$ such that*

- *$\mathcal{A}_H = (\Sigma, Q_H, Q_{0,H}, \delta_H, F_H, C_H)$ is a weighted automaton,*
- *$\tau_H : Q_H \mapsto 2^{Q_M \times \Sigma \times Q_M}$,*
- *$\Gamma_H : Q_H \mapsto \mathbb{N}^{Q_M \times \Sigma \times Q_M}$,*
- *$\mathcal{A}_H$ has an initial $q_I \in Q_{0,H}$, an* idle state*, such that $\tau_H[q_I] = \delta_M$, $\Gamma_H[q_I] = C_M$ and for every $a \in \Sigma$, $\mathcal{A}_H$ has a transition (q_I, a, q_I) of weight 0.*

The functions τ_H, Γ_H determine the transition relation and cost function for $\mathcal{A}_M$ at each step. Intuitively, they should encode modifications applied to the transition relation and cost function of $\mathcal{A}_M$ rather than their complete descriptions. E.g. blind a-transitions $\tau_H[q_a] = \{(q, b, q') : (q, a, q') \in \delta_M, b \in \Sigma\}$, i.e., the automaton moves as it would read a, regardless of the actual letter. Having δ_M, $\tau_H[q_a]$ can be simply defined regardless of complexity of δ_M.

Let $\mathcal{A}_M = (\Sigma, Q_M, Q_{0,M}, \delta_M, F_M, C_M)$ be a weighted automaton. For a hypervisor $H = (\mathcal{A}_H, \tau_H, \Gamma_H)$ with $\mathcal{A}_H = (\Sigma, Q_H, Q_{0,H}, \delta_H, F_H, C_H)$, the *semi-direct product* $\mathcal{A}_M \ltimes H$ is a weighted generalized Büchi automaton $(\Sigma, Q_H \times Q_M, Q_{0,H} \times Q_{0,M}, \delta, C, \{F_1, F_2\})$ defined as follows:

- $\delta = \{(\langle q_1, q_2\rangle, a, \langle q_1', q_2'\rangle) : a \in \Sigma, (q_1, a, q_1') \in \delta_H, (q_2, a, q_2') \in \tau_H[q_1]\}$,
- $F_1 = F_H \times Q_M$ and $F_2 = Q_H \times F_M$, that is the automaton should visit infinitely often accepting states of $\mathcal{A}_M$ and those of $\mathcal{A}_H$,
- $C(\langle q_1, q_2\rangle, a, \langle q_1', q_2'\rangle) = C_H(q_1, a, q_1') + \Gamma_H[q_1](q_2, a, q_2')$.

Observe that for every hypervisor H, and every $\mathcal{A}_M$ recognizing the set of traces of M, $\mathcal{A}_M \ltimes H$ defines an automatic similarity measure related to M. Indeed, due to existence of the idle state, the automaton $\mathcal{A}_M \ltimes H$ can just simulate $\mathcal{A}_M$, therefore for every trace of M, $\mathcal{A}_M \ltimes H(w) = 0$. Conversely, the hypervisor method is complete, i.e., every automatic similarity measure d_M is computed by an automaton which is the semi-product of $\mathcal{A}_M$ and some H.

Let d_M be an automatic similarity measure and let $\mathcal{A}_{dist}$ be an automaton computing it. Consider a hypervisor $H = (\mathcal{A}_H, \tau_H, \Gamma_H)$ such that $\mathcal{A}_H$ can either begin in q_I and stay there forever, or it can begin in $q_{0,dist}$, simulate the execution of $\mathcal{A}_{dist}$, and neglect the automaton $\mathcal{A}_M$, i.e., for every $q \in Q_H \setminus \{q_I\}$, $\tau_H[q]$ is the full relation and $\Gamma_H[q]$ is always 0. Then, for every w, $\mathcal{A}_M \ltimes H(w) = \mathcal{A}_{dist}(w)$, therefore $\mathcal{A}_M \ltimes H$ computes d_M.

However, this is a degenerate case. We rather focus on showing that the hypervisor-based approach is a convenient and reasonably uniform (w.r.t. $\mathcal{A}_M$) way of modeling similarity measures. In the following we shall give several examples supporting this thesis.

Observe that using aforementioned blind a-transitions one can simply define a similarity measure $d_M(w) = \inf\{vRw : v \text{ is a trace of } M\}$ based on an automatic weighted relation R. We leave that as an exercise for the reader.

Since the semi-direct product of a weighted automaton $\mathcal{A}_M$ and H is again a weighted automaton, this construction can be iterated $((\mathcal{A}_M \ltimes H) \ltimes H')$. Although iteration can be avoided (Lemma 19), iterated definitions are often simpler (cf. Example 20).

Lemma 19. *Let $\mathcal{A}$ be a weighted automaton and H_1, H_2 be hypervisors for $\mathcal{A}$ and $\mathcal{A} \ltimes H_1$. There effectively exists a hypervisor H_3 such that for every w, $(\mathcal{A} \ltimes H_1) \ltimes H_2(w) = \mathcal{A} \ltimes H_3(w)$.*

Example 20. **(Edit distance)** We define the hypervisor $Del = (\mathcal{A}_{Del}, \tau_{Del}, \Gamma_{Del})$ computing deletions of letters. The automaton $\mathcal{A}_{Del}$ has two states, the idle state q_I, and the state q_D responsible for deletion. In the deletion state q_D, the automaton $\mathcal{A}_M$ ignores the input letter and remains in its current state, i.e., $\tau_{Del}[q_D] = \{(q, a, q) : q \in Q, a \in \Sigma\}$. The cost functions $\Gamma_H[q_I], \Gamma_H[q_L]$ assign 0 weight to every transition of $\mathcal{A}_M$. Both, q_I, q_D, are initial states in $\mathcal{A}_H$ and δ_H is the full relation, i.e., $\delta_H = \{(q_1, a, q_2) : q_1, q_2 \in Q_H, a \in \Sigma\}$. Transitions from q_I have weight 0 (in $\mathcal{A}_H$), whereas those from q_D have weight 1. Clearly, $\mathcal{A}_M \ltimes Del$ computes a similarity measure such that the weight of an ω-word w is the least number of deletions necessary to transform w to a trace of M. $\mathcal{A}_M \ltimes Del$ extends from ω-words to transition systems by taking supremum over all traces of a transition system.

Similarly, one can define hypervisors Ins, Sub, Tra computing insertions, a single letter substitutions or transpositions of adjacent letters necessary to transform a given ω-word to an ω-word accepted by the hypervised automaton. Then,

the automaton $\mathcal{A}_{edit}$, defined as $(((M \ltimes Del) \ltimes Sub) \ltimes Tra) \ltimes Ins$, computes the edit distance between w and the set of traces of M. Indeed, if v, a trace of M, can be obtained from w by applying deletions, substitution, transpositions and insertions, it can be obtained by applying the same number of these operations in precisely that order, i.e., first deletions, next substitutions etc.

Example 21. **(An unreliable channel)** Consider the similarity measure d_M from Example 8. Assume that those parties, P_1, P_2, communicate through a sheared variable a. Suppose that the transition relation for M is given symbolically by a propositional formula $\mathcal{N}(\boldsymbol{p_1 p_2 a}, \boldsymbol{p'_1 p'_2 a'})$, where $\boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{a}$ are vectors of propositional variables that represent the current state of P_1, P_2 and a, and $\boldsymbol{p'_1}, \boldsymbol{p'_2}, \boldsymbol{a'}$ represent their next state. All transitions in M have weight 0.

Consider a hypervisor $H = (\mathcal{A}_H, \tau_H, \Gamma_H)$ such that $\mathcal{A}_H$ has two states: the idle state q_I, and q_L, the state of a packet being lost. The cost functions $\Gamma_H[q_I], \Gamma_H[q_L]$ assign 0 weight to every transition, which can be easily expressed symbolically. Then $\tau[q_I], \tau[q_L]$ are represented by $\mathcal{N}$, and $\mathcal{N}_L(\boldsymbol{p_1 p_2 a}, \boldsymbol{p'_1 p'_2 a'}) \equiv \exists \boldsymbol{a''}(\mathcal{N}(\boldsymbol{p_1 p_2 a}, \boldsymbol{p'_1 p'_2 a''}) \wedge \boldsymbol{a} = \boldsymbol{a'})$. Thus, when $\mathcal{A}_H$ is in the state q_D, $\mathcal{A}_M$ executes the usual transition, but immediately after that a is being reset to its previous value. Clearly, $\mathcal{A}_M \ltimes H$ defines the similarity measure from Example 8.

Now, by employing different weighting schemes, we can ask a whole range of questions. For Sum weighting scheme, the stability radius is the maximal number of lost packets tolerated by the system, whereas for LimAvg weighting scheme it gives the maximal *average ratio* of lost packets tolerated by the system.

Example 22. **(Active environment)** We can extend the idea from Example 21 to many processes where content of packets may be altered during communication. It is possible, as in the Dolev-Yao model for verification of cryptographic protocols [12], to simulate a scenario where all communication channels are controlled by the *intruder* who can intercept and forge packets. As it is unlikely that the system is immune to arbitrary actions of the intruder, the model-measure tells us *how* vulnerable the system is. E.g. the system works correctly as long as no more than 5 packets are forged.

The hypervisor approach can be straightforwardly adapted to the branching-time case. A (tree) hypervisor H is a triple $(\mathcal{A}_H, \tau_H, \Gamma_H)$ such that $\mathcal{A}_H$ is a weighted automaton over ω-trees with varying (but bounded) degree and τ_H, Γ_H associate with each state of $\mathcal{A}_H$ a (tree) transition relation and cost function.

Now, we shall present examples of branching-time similarity measures. The first example is a class of measures inherited from the linear-time case. In the linear-time case, $d_M(M')$ is defined as the supremum over weights of all traces of M', therefore linear-time similarity measures naturally translate to branching-time similarity measures over Sup weighting schemes. Indeed, consider an ω-word automaton $\mathcal{A}^w_M$. By extending the labeling of M to $\Sigma \times Q$, we can assume that $\mathcal{A}^w_M$ is deterministic. Then, it can be transformed to an ω-tree automaton $\mathcal{A}^t_M$ that accepts precisely those ω-trees whose paths are accepted by $\mathcal{A}^w_M$. This idea can be generalized to weighted tree automata over Sup weighting schemes to get the following:

Proposition 23. *Let M be a transition system and let $\mathcal{A}^w_M, \mathcal{A}^t_M$ be a word and tree automata representing M. Every word hypervisor H^w can be translated to a tree hypervisor H^t such that the similarity measures defined by $\mathcal{A}^w_M \ltimes H^w$ and by $\mathcal{A}^t_M \ltimes H^t$ agree on every transition system M' labeled by $\Sigma \times Q^w_H \times Q^w_M$.*

In particular, Examples 20, 21, 22 can be adapted to the branching-time case.

Another feature of tree hypervisors, which is incompatible with the linear-time case, is the ability to clone (or prune) a transition. Transition cloning at a state can be easily implemented as follows. For every $j \in \{1, \ldots, k\}$, the hypervisor has a cloning state $q_{c,j}$ such that $\tau_H[q_{c,j}]$ changes $q_0 \xrightarrow{a} \langle q_1, \ldots, q_k \rangle$, an original transition of $\mathcal{A}_M$, to $q_0 \xrightarrow{a} \langle q_1, \ldots, q_k, q_j \rangle$. In a similar way one can define transition pruning. By combining cloning and pruning one can implement the robustness notion from [14]. Indeed, the language of all execution trees of $M \parallel M'$, where branching degree of M' is bounded by B, can be obtained by the combination of transition cloning (where each transition is cloned at most B times), and arbitrary pruning. Thus, robustness of open systems defined in [14] is a special case of model measuring.

Example 24. **(Mutations)** Removal of behaviors according to [17] is a special case of our transition pruning. Generally, mutations that modify or add behaviors can be straightforwardly implemented using the hypervisor approach. Thus, all mutations considered in [17] can be expressed by similarity measures.

Finally, the model-measuring problem subsumes the robustness distance [3]:

Proposition 25. *Let M be a transition system. There (effectively) exists a similarity measure d_M such that for every transition system M', the value of the robustness distance from M to M' equals to $1 - [M']_{d_M}$.*

5 Conclusions

We have defined the model-measuring problem, which generalizes several previously studied notions of robustness in verification. We have shown a way to express several distances (edit distance; semantic distance: the number of lost packets; etc.) in a convenient way, based on weighted automata, which admits a succinct symbolic representation.

The algorithms computing the model measure follow the same basic scheme as standard automata-based model-checking algorithms. This suggests that our method can be implemented on the basis of existing model-checking tools.

The model-measuring problem can be extended to the real-time case. It remains to construct a variety of similarity measures in the timed case.

References

1. Baader, F., Peñaloza, R.: Automata-based axiom pinpointing. J. Autom. Reasoning 45(2), 91–129 (2010)
2. Boker, U., Chatterjee, K., Henzinger, T.A., Kupferman, O.: Temporal specifications with accumulative values. In: LICS, pp. 43–52. IEEE Computer Society (2011)

3. Cerný, P., Henzinger, T.A., Radhakrishna, A.: Simulation distances. Theor. Comput. Sci. 413(1), 21–35 (2012)
4. Chatterjee, K., Doyen, L.: Energy parity games. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010, Part II. LNCS, vol. 6199, pp. 599–610. Springer, Heidelberg (2010)
5. Chatterjee, K., Doyen, L., Edelsbrunner, H., Henzinger, T.A., Rannou, P.: Mean-payoff automaton expressions. CoRR, abs/1006.1492 (2010)
6. Chatterjee, K., Doyen, L., Henzinger, T.A.: Quantitative languages. In: CSL, pp. 385–400 (2008)
7. Chatterjee, K., Doyen, L., Henzinger, T.A.: Alternating weighted automata. In: Kutyłowski, M., Charatonik, W., Gębala, M. (eds.) FCT 2009. LNCS, vol. 5699, pp. 3–13. Springer, Heidelberg (2009)
8. Chatterjee, K., Henzinger, T.A., Jobstmann, B., Singh, R.: Measuring and synthesizing systems in probabilistic environments. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 380–395. Springer, Heidelberg (2010)
9. Chatterjee, K., Henzinger, T.A., Jurdzinski, M.: Mean-payoff parity games. In: LICS, pp. 178–187. IEEE Computer Society (2005)
10. Chockler, H., Kupferman, O., Vardi, M.Y.: Coverage metrics for temporal logic model checking. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 528–542. Springer, Heidelberg (2001)
11. Das, S., Banerjee, A., Basu, P., Dasgupta, P., Chakrabarti, P.P., Mohan, C.R., Fix, L.: Formal methods for analyzing the completeness of an assertion suite against a high-level fault model. In: VLSI Design, pp. 201–206. IEEE Computer Society (2005)
12. Dolev, D., Yao, A.C.: On the security of public key protocols. In: FOCS 1981, pp. 350–357. IEEE Computer Society, Washington, DC (1981)
13. Filar, J., Vrieze, K.: Competitive Markov decision processes. Springer-Verlag New York, Inc., New York (1996)
14. Grumberg, O., Long, D.E.: Model checking and modular verification. In: Groote, J.F., Baeten, J.C.M. (eds.) CONCUR 1991. LNCS, vol. 527, pp. 250–265. Springer, Heidelberg (1991)
15. Henzinger, T.A.: From boolean to quantitative notions of correctness. In: POPL 2010, pp. 157–158. ACM, New York (2010)
16. Hinrichsen, D., Son, N.K.: Stability radii of linear discrete-time systems and symplectic pencils. Int. J. of Robust and Nonlinear Control 1(2), 79–97 (1991)
17. Kupferman, O., Li, W., Seshia, S.A.: A theory of mutations with applications to vacuity, coverage, and fault tolerance. In: FMCAD, pp. 1–9. IEEE (2008)
18. Kupferman, O., Vardi, M.Y.: Robust satisfaction. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 383–398. Springer, Heidelberg (1999)
19. Kupferman, O., Vardi, M.Y.: Vacuity detection in temporal model checking. STTT 4(2), 224–233 (2003)
20. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. J. ACM 47(2), 312–360 (2000)
21. Leung, H.: Limitedness theorem on finite automata with distance functions: an algebraic proof. Theor. Comput. Sci. 81(1), 137–145 (1991)
22. Mohri, M.: Semiring frameworks and algorithms for shortest-distance problems. Journal of Automata, Languages and Combinatorics 7(3), 321–350 (2002)
23. Zwick, U., Paterson, M.: The complexity of mean payoff games on graphs. Theor. Comput. Sci. 158(1&2), 343–359 (1996)

Safety Verification of Asynchronous Pushdown Systems with Shaped Stacks

Jonathan Kochems and C.-H. Luke Ong

University of Oxford

Abstract. In this paper, we study the program-point reachability problem of concurrent pushdown systems that communicate via unbounded and unordered message buffers. Our goal is to relax the common restriction that messages can only be retrieved by a pushdown process when its stack is empty. We use the notion of partially commutative context-free grammars to describe a new class of asynchronously communicating pushdown systems with a mild shape constraint on the stacks for which the program-point coverability problem remains decidable. Stacks that fit the shape constraint may reach arbitrary heights; further a process may execute any communication action (be it process creation, message send or retrieval) whether or not its stack is empty. This class extends previous computational models studied in the context of asynchronous programs, and enables the safety verification of a large class of message passing programs.

Keywords: Pushdown systems, asynchronous message passing, verification.

1 Introduction

The safety verification of concurrent and distributed systems, such as client-server environments, peer-to-peer networks and the myriad web-based applications, is an important topic of research. We consider *asynchronously communicating pushdown systems* (ACPS), a model of computation for such systems suitable for the algorithmic analysis of the reachability problem. Each process of the model is a pushdown system; processes may be spawned dynamically and they communicate asynchronously via a number of unbounded message buffers which may be ordered or unordered. In order to obtain a decision procedure for reachability, some models restrict the retrieval (or, dually, the sending) of messages or the scheduling of tasks, allowing it to take place only when the call stack is empty.

Can these restrictions on call stacks be relaxed? Unfortunately[1] some form of constraint on the call stacks in relation to the communication actions is unavoidable. Inspired by the work on asynchronous procedure calls [30, 22, 17], we consider processes that communicate asynchronously via a fixed number of unbounded and unordered message buffers which we call channels. Because channels are unordered, processes cannot observe the precise sequencing of such concurrency actions as message send and process creation; however, the sequencing of other actions, notably blocking actions such as message retrieval which requires synchronisation, is observable. If the behaviour of a

[1] Any analysis that is both context-sensitive and synchronisation-sensitive is undecidable [29].

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 288–302, 2013.

process is given by its action sequences, then we may postulate that certain actions *commute* with each other (over sequential composition) while others do not. To formalise these assumptions, we make use of *partially commutative context-free grammars* (PCCFG) [7], introduced recently by Czerwinski et al. as a study in process algebra. A PCCFG is just a context-free grammar equipped with an irreflexive symmetric relation, called *independence*, over an alphabet Σ of terminal symbols, which precisely captures the symbols that *commute* with each other. In our model, a process is described by a PCCFG that generates the set of its action sequences; terminal symbols represent concurrency and communication actions, while the non-terminal symbols represent procedure calls; and there is an induced notion of commutative procedure calls. With a view to deciding reachability, a key innovation of our work is to summarise the effects of the commutative procedure calls on the call stack. Rather than keeping track of the contents of the stack, we precompute the actions of those procedure calls that produce only commutative side-effects, and store them in caches on the call stack. The non-commutative procedure calls, which are left on the stack *in situ*, act as separators for the caches of commutative actions. As soon as the top non-commutative non-terminal on the stack is popped, which may be triggered by a concurrency action, the cache just below it is unlocked, and all the cached concurrency actions are then despatched at once.

In order to obtain a decision procedure for (a form of reachability called) *coverability*, we place a natural constraint on the shape of call stacks: at all times, no more than an *a priori* fixed number of *non-commutative non-terminals* may reside in the stack. Note that because the constraint does not apply to commutative non-terminals, call stacks can grow to arbitrary heights. Thanks to the shape constraint, we can prove that the coverability problem is decidable by an encoding into well-structured transition systems. To our knowledge, this class extends previous computational models studied in the context of asynchronous programs. Though our shape constraint is semantic, we give a simple sufficient condition which is expressed syntactically, thus enabling the safety verification of a large class of message-passing programs.

Example 1. In Figures 1 and 2 we give an example program written in a version of Erlang that employs channels (as opposed to Actor-style mailboxes), implementing a simple replicated workers pattern. It consists of a distributor process that initially spawns a number of workers, sets up a single shared resource, and distributes one task per worker over a one-to-many channel. Each worker runs a task-processing loop. Upon reception of a task, the worker recursively decomposes it, which involves communicating with the shared resource at each step. Note that the communication of each worker with the resource is protected by a lock. For the worker, the decomposition has two possible outcomes: (i) the task is partially solved, generating one subtask and an intermediate result or (ii) the task is broken down into one subtask and one new distributable task. In case (i) the worker recursively solves the subtask and combines the result with the intermediate result. In case (ii) the worker recursively solves the subtask and subsequently dispatches the newly generated distributable task before returning. When a worker has finished processing a task, it relays the result to the server and awaits a new task to process. We have left the implemention of the functions decompose_task and combine

```
1  main() → setup_network(),
2           redistribute () .
3
4  setup_network() →
5    spawn(worker),
6    case (∗) of
7      true  → setup_network();
8      false →
9        spawn(res_start(init )),
10       toResource ! isReady,
11       receive toDistributor :
12         ready → ()
13       end;
14   end, toWorkers ! task.
15
16 redistribute () →
17   receive toDistributor :
18     redist (Task) → toWorkers ! Task;
19     result (Result) → print (Result);
20   end, redistribute () .
```

```
21 % Resource
22 res_start (S) =
23   fun() → toDistributor ! ready,
24           resource(S)
25   end.
26 resource(S) →
27   receive toResource:
28     lock_req →
29       toWorkers ! locked,
30       resource_locked(S)
31   end.
32
33 resource_locked(S) →
34   receive toResource:
35     unlock_req → resource(S);
36     getState →
37       toWorkers ! state(S),
38       resource_locked(S);
39     update(X) → resource_locked(X)
40   end.
```

Fig. 1. A resource and a task distributor

open; for the purpose of this example we only assume that they do not perform any concurrency actions, but they may be recursive functions.

Note that the call stacks of both the distributor and the workers may reach arbitrary heights, and communication actions may be performed by a process at any stage of the computation, regardless of stack height. For example the worker sends and receives messages at every decomposition, and each recursive call increases the height of the call stack.

An interesting verification question for this example program is whether the locking mechanism for the shared resource guarantees exclusive access to the shared resource for each worker process in its critical section.

A Computational Model. To verify programs such as the above we need a computational model that allows us to model recursive procedure calls, message passing concurrency actions and process creation. Once the obvious abstractions are applied to make the data and message space finite, we arrive at a network of pushdown systems (equivalently context-free grammars) which can communicate asynchronously over a finite number of channels with unbounded capacity. Since we are interested in a class of such systems with decidable verification problems we assume that channels are unordered (FIFO queues with finite control are already Turing powerful [5]).

Outline. The rest of the paper is organised as follows. In Section 2 we present our model of asynchronous partially commutative pushdown systems (APCPS), its (standard) semantics and a verification problem. In Section 3 we investigate an alternative semantics for APCPS, a corresponding verification problem, and relate it to the verification problem of Section 2. In Section 4 we introduce the class of APCPS with shaped stacks and show that the verification problems are decidable for this class. In Section 5 we discuss related work and then conclude.Owing to space constraints proofs are omitted and can be found in the long version of the paper [24].

```
worker() →
  receive toWorkers:
    Task →
      result = do_task(Task),
      toDistributor ! result ;
  end, worker().

do_task(Task) →
  case decompose(Task) of
    local (Task', Int_result ) →
      Result = do_task(Task'),
      Result' =
        combine(Result,Int_result)
      return Result';
    redist (Task', Task'') →
      Result = do_task(Task'),
      toDistributor ! Redist(Task'') ,
      return Result;
  end.

combine(res,res') → ...

decompose(Task) →
  lock(toResource),
  toResource ! getState,
  ?label("critical "),
  receive toWorkers:
    state(State) →
      (Result,Update) =
        decompose_task(Task, State)
  end,
  toResource ! update(Update),
  unlock(toResource),
  return Result.

lock(C) →
  C ! lock_req,
  receive toWorkers:
    locked → ()
  end.
unlock(C) → C ! unlock_req.

decompose_task(Task,State) →...
```

Fig. 2. A worker that recursively solves tasks and shares its workload

Notation. We write $\mathbb{M}[U]$ for the set of multisets over the set U, and we use $[\cdot]$ to denote multisets explicitly e.g. we write $[u, u, v, v]$ to mean the multiset containing two occurrences each of u and v. Given multisets M_1 and M_2, we write $M_1 \oplus M_2$ for the multiset union of M_1 and M_2. We write U^* for the set of finite sequences over U, and let $\alpha, \beta, \gamma, \mu, \nu, \ldots$ range over U^*. We define the *Parikh image* of $\alpha \in U^*$ to be the multiset over U, $\mathbb{M}_U(\alpha) : u \mapsto |\{i \mid \alpha(i) = u\}|$; we drop the subscript and write $\mathbb{M}(\alpha)$ whenever it is clear from the context. We order multisets in the usual way: $M_1 \leq_{\mathbb{M}} M_2$ just if for all u, $M_1(u) \leq M_2(u)$. Let $M \in \mathbb{M}[U]$ and $U_0 \subseteq U$. We define $M \restriction U_0$ to be the multiset M restricted to U_0 i.e. $(M \restriction U_0) : u \mapsto M(u)$ if $u \in U_0$, and 0 otherwise. We write $U \uplus V$ for the disjoint union of sets U and V.

2 Asynchronous Communicating Pushdown Systems

In this section we introduce our model of concurrency, *asynchronous partially commutative pushdown systems*. Processes are modelled by a variant of context-free grammars, which distinguish commutative and non-commutative concurrency actions. Communication between processes is asynchronous, via a fixed number of unbounded and unordered message buffers, which we call *channels*.

Preliminaries. An *independence relation* I over a set U is a symmetric irreflexive relation over U. It induces a congruence relation $\simeq_I$ on U^* defined as the least equivalence relation R containing I and satisfying: $(\mu, \mu') \in R \;\Rightarrow\; \forall \nu_0, \nu_1 \in U^* : (\nu_0 \,\mu\, \nu_1, \nu_0 \,\mu'\, \nu_1) \in R$.

Let I be an independence relation over U. An element $a \in U$ is *non-commutative* (with respect to I) just if $\forall b \in U : (a, b) \notin I$ i.e. a does not commute with any other element. An element b is *commutative* (with respect to I) just if for each $c \in U$, if c

is not non-commutative then $(c, b) \in I$; intuitively it means that b commutes with all elements of U except those that are non-commutative. We call an independence relation I *unambiguous* if just every element of U is either commutative or non-commutative.

Definition 1. Let Σ be an alphabet of terminal symbols and $I \subseteq \Sigma \times \Sigma$ an independence relation over Σ. A *partially commutative context-free grammar* (PCCFG) is a quintuple $\mathcal{G} = (\Sigma, I, \mathcal{N}, \mathcal{R}, S)$ where $S \in \mathcal{N}$ is a distinguished start symbol, and $\mathcal{R}$ is a set of rewrite rules of the following types:[2] let $A \in \mathcal{N}$ (i) $A \to a$ where $a \in \Sigma \cup \{\epsilon\}$, (ii) $A \to a\,B$ where $a \in \Sigma$, $B \in \mathcal{N}$, (iii) $A \to B\,C$ where $B, C \in \mathcal{N}$. We refer to each $\rho \in \mathcal{R}$ as a $\mathcal{G}$*-rule*.

The (leftmost) derivation relation $\to_{\text{seq}}$ is a binary relation over $(\Sigma \cup \mathcal{N})^*/\!\simeq_I$ defined as $X\,\alpha \to_{\text{seq}} \beta\,\alpha$ if $X \to \beta$ is a $\mathcal{G}$-rule. Note the derivation relation is defined over the quotient by $\simeq_I$, so the words generated are congruence classes induced by $\simeq_I$. As usual we denote the n-step relation as $\to^n_{\text{seq}}$ and reflexive, transitive closure as $\to^*_{\text{seq}}$.

We further define a k-index derivation to be a derivation in which every term contains at most k occurrences of non-terminals. Recent work [14, 12] has shown that for every commutative context-free grammar $\mathcal{G}$ there exists $k \geq 1$ such that the entire language of $\mathcal{G}$ can be generated by derivations of index k.

PCCFG was introduced by Czerwinski et al. as a study in process algebra. They investigated [7] the decidability of bisimulation for a class of processes described by PCCFG where the commutativity of the sequential composition is constrained by an independence relation on non-terminals. We propose to use words generated by PCCFGs to represent the sequence of concurrency actions of processes.

2.1 Asynchronous Partially Commutative Pushdown Systems

Our model of computation, asynchronous partially commutative pushdown systems, are in essence PCCFGs equipped with an independence relation over an alphabet Σ of terminal symbols, which represent the concurrency actions and program point labels. First some notation. Let Chan be a finite set of *channel names* ranged over by c, Msg be a finite *message alphabet* ranged over by m, and $\mathcal{L}$ be a finite set of *program point labels* ranged over by l, l', l_1, etc. Further let $\mathcal{N}$ be a finite set of non-terminal symbols. We derive an alphabet Σ of terminal symbols

$$\Sigma \;:=\; \mathcal{L} \cup \{c\,!\,m, c\,?\,m \mid c \in \mathit{Chan}, m \in \mathit{Msg}\} \cup \{\nu X \mid X \in \mathcal{N}\}. \qquad (1)$$

An action of the form $c\,!\,m$ denotes the sending of the message m to channel c, $c\,?\,m$ denotes the retrieval of message m from channel c, and νX denotes the spawning of a new process that begins execution from X. We will use a, a', b, etc. to range over Σ. Our computational model will emit program point labels in its computation, allowing us to pose questions of reachability. We will now define the computational power of our processes in terms of PCCFGs.

[2] Identifying rules of type (ii), which is a special case of type (iii), allows us to distinguish tail-recursive and non-tail recursive calls, which will be handled differently in the sequel, beginning with Definition 4.

The words that are generated by a process *qua* PCCFG represent its action sequences. Because channels are unordered, processes will not be able to observe the precise sequencing of concurrency actions such as message send and process creation; however the sequencing of other actions such as message retrieval is observable. Using the language of partially commutative context-free grammar, we can make this sensitivity to sequencing precise by an independence relation on actions.

An Independence Relation for the Concurrency Actions. Let $\Xi \subseteq \Sigma$, we define the independence relation over Σ generated by Ξ as

$$IndRel_{\Sigma}(\Xi) := \{(a, a'), (a', a) \mid a, a' \in \Xi, a \neq a'\}$$

Now let $\Sigma^{\flat} := \mathcal{L} \cup \{c\,!\,m \mid c \in Chan, m \in Msg\} \cup \{\nu X \mid X \in \mathcal{N}\}$ be the subset of Σ consisting of the program point labels and the send and spawn actions. It is straightforward to see that $IndRel_{\Sigma}(\Sigma^{\flat})$ is, by construction, an unambiguous independence relation over Σ. Thus $IndRel_{\Sigma}(\Sigma^{\flat})$ allows us to commute all concurrency actions *except* receive. Further we allow program point labels to commute. This is harmless, since our goal is to analyse (a form of) control-state reachability, i.e. the question whether a particular label can be reached, as opposed to questions that require sequential reasoning such as whether label l_1 will be reached before l_2 is reached.

We can now lift the independence relation to the non-terminals of a PCCFG $\mathcal{G}$. Let I be the least subset of $(\mathcal{N} \cup \Sigma)^2$ such that (i) $IndRel_{\Sigma}(\Sigma^{\flat}) \subseteq I$, and (ii) for all $b \in \Sigma \cup \mathcal{N}$ and $A \in \mathcal{N}$, if $\forall a \in \mathrm{RHS}(A) : (a, b) \in I$ then $\{(A, b), (b, A)\} \subseteq I$, where $\mathrm{RHS}(A) := \{a \in \mathcal{N} \cup \Sigma \mid A \to \alpha \in \mathcal{G}, a \text{ occurs in } \alpha\}$. We note that I, which is well-defined, is an unambiguous independence relation over $\mathcal{N} \cup \Sigma$. Thus we can partition both Σ and $\mathcal{N}$ into Σ^{com} and $\mathcal{N}^{\mathrm{com}}$, the *commutative* actions and non-terminals respectively, and $\Sigma^{\neg\mathrm{com}}$ and $\mathcal{N}^{\neg\mathrm{com}}$ their *non-commutative* counterparts respectively.

We can now define our model of computation.

Definition 2. Assume $\mathcal{L}$, $Chan$, Msg and $\mathcal{N}$ as introduced earlier, and the derived alphabet Σ of terminals as defined in (1). An *asynchronous partially commutative pushdown system* (APCPS) is just a PCCFG $\mathcal{G} = (\Sigma, I, \mathcal{N}, \mathcal{R}, S)$.

Henceforth we fix $\mathcal{L}$, $Chan$, Msg and $\mathcal{N}$, and the derived (1) alphabet Σ of terminals.

2.2 Standard Semantics

The operational semantics is given as a transition system. A configuration of the system is a pair, consisting of a parallel composition of processes and a set of channels. We represent the state of a single process as an element of $Control := (\Sigma \cup \mathcal{N})^*/_{\simeq_I}$. The derivation relation of PCCFGs, $\to_{\mathrm{seq}}$, defines how processes make *sequential* transitions. Processes interact concurrently by message passing via a fixed set of unbounded and unordered channels.

Definition 3 (Standard Concurrent Semantics). The *configurations* are elements of $\mathbb{M}[Control] \times (Chan \to \mathbb{M}[Msg])$. For simplicity, we write a configuration (say) $([\alpha, \beta], \{c_1 \mapsto [m_a, m_b, m_b], c_2 \mapsto []\})$ as $\alpha \parallel \beta \lhd [m_a, m_b, m_b]^{c_1}, []^{c_2}$. We abbreviate a set of processes running in parallel as Π and a set of channels by Γ with names

in $Chan$. The operational semantics for APCPS, a binary relation $\rightarrow_{\text{con}}$ over configurations, is then defined by induction over the rule:

$$\frac{\alpha \rightarrow_{\text{seq}} \alpha'}{\alpha \parallel \Pi \lhd \Gamma \rightarrow_{\text{con}} \alpha' \parallel \Pi \lhd \Gamma} \tag{2}$$

and the following axioms: let $m \in Msg, c \in Chan, l \in \mathcal{L}$ and $X \in \mathcal{N}$

$$(c\,?\,m)\,\alpha \parallel \Pi \lhd ([m] \oplus q)^c, \Gamma \rightarrow_{\text{con}} \alpha \parallel \Pi \lhd q^c, \Gamma \tag{3}$$

$$(c\,!\,m)\,\alpha \parallel \Pi \lhd q^c, \Gamma \rightarrow_{\text{con}} \alpha \parallel \Pi \lhd ([m] \oplus q)^c, \Gamma \tag{4}$$

$$l\,\alpha \parallel \Pi \lhd \Gamma \rightarrow_{\text{con}} \alpha \parallel \Pi \lhd \Gamma \tag{5}$$

$$(\nu X)\,\alpha \parallel \Pi \lhd \Gamma \rightarrow_{\text{con}} \alpha \parallel X \parallel \Pi \lhd \Gamma. \tag{6}$$

The *start configuration* is $S \lhd \emptyset$. We define a partial order on configurations: $\Pi \lhd \Gamma \leq \Pi' \lhd \Gamma'$ just if $\Pi \leq_{\mathbb{M}} \Pi'$ and for every $c \in Chan$, $\Gamma(c) \leq_{\mathbb{M}} \Gamma'(c)$.

2.3 Program-Point Coverability

In the sequential setting of (ordinary) pushdown systems, the control-state reachability problem is of central interest. In our notation, it asks, given a control-state A, if it is possible to reach a process-configuration $A\,\alpha$ where A is the control-state and α is some call stack. It should be clear that an equivalent problem is to ask whether $l\,\alpha$ is reachable, where l is a program-point label. We prefer a formulation that uses program-point labels because it simplifies our argument (and is equi-expressive).

In the concurrent setting, we wish to know whether, given an APCPS and program-point labels $l_1, \ldots, l_n$, there exist call stacks $\alpha_1, \ldots, \alpha_n$ and channel contents Γ such that the configuration $l_1\,\alpha_1 \parallel \cdots \parallel l_n\,\alpha_n \lhd \Gamma$ is $\rightarrow_{\text{con}}$-reachable, possibly in parallel with some other processes. Note that this question allows us to express not just control-state reachability queries but also mutual exclusion properties. We state the problem of program-point coverability more formally as follows.

Verification Problem 1 (Program-Point Coverability). Given an APCPS $\mathcal{G}$ and program point labels $l_1, \ldots, l_n$, a tuple $(\mathcal{G}; l_1, \ldots, l_n)$ is a yes-instance of the *program-point coverability* problem just if there exist a configuration $\Pi \lhd \Gamma$ and $\alpha_1, \ldots, \alpha_n \in (\Sigma \cup \mathcal{N})^*/_{\simeq_I}$ such that $\Pi \lhd \Gamma$ is $\rightarrow_{\text{con}}$-reachable and $l_1\alpha_1 || \cdots || l_n\alpha_n \lhd \emptyset \leq \Pi \lhd \Gamma$.

The program-point coverability problem allows us to characterise "bad-configurations" c_{bad} in terms of program-point labels. We regard a configuration c that covers c_{bad}, in the sense that ($c_{\text{bad}} \leq c$), also as "bad". Using program-point coverability, we can express whether any such configuration is reachable

Example 2. Consider the program in Figures 1 and 2 and call it P. The problem of whether each worker has exclusive access to the shared resource in its critical section is expressible as a program-point coverability problem. A bad configuration is one in which two worker processes are executing the line marked by `?label(" critical ")`. We can thus see that $(P$;`?label(" critical ")`,`?label(" critical ")`$)$ is an instance of the program-point coverability problem; a no answer implies mutual exclusion, a yes answer tells us that two worker processes can be simultaneously within their critical section.

The program-point coverability problem is undecidable for unconstrained APCPS. In fact APCPS is Turing powerful: it is straightforward to simulate a system with two synchronising pushdown systems.

3 An Alternative Semantics for APCPS

In this section we present an alternative semantics for APCPS which captures enough information to solve the program-point coverability problem. The key idea is to summarise the effects of the commutative non-terminals. In the alternative semantics, rather than keeping track of the contents of the call stack, we precompute the actions of those procedure calls that produce only *commutative* side-effects, i.e. sends, spawns and program point labels, and store them in caches on the call stack. The non-commutative procedure calls, which are left on the call stack, then act as separators for the caches of commutative side-effects. As soon as the top non-commutative non-terminal on the stack is popped, which may be triggered by a concurrency action, the cache just below it is unlocked. The cached actions are made effective instantaneously. This is enough to ensure a precise correspondence between the program-point coverability problem for APCPS and a corresponding coverability problem for our alternative semantics.

An Alternative Semantics. First we introduce a representation of the states of a process. Let $k \in \mathbb{N} \cup \{\infty\}$.

$$
\begin{aligned}
\mathit{TermCache} &:= \mathbb{M}[\Sigma^{\text{com}}] \qquad & \mathit{MixedCache} &:= \mathbb{M}[\Sigma^{\text{com}} \cup \mathcal{N}^{\text{com}}] \\
\mathit{NonTermCache} &:= \mathbb{M}[\mathcal{N}^{\text{com}}] \qquad & \mathit{Cache} &:= \mathit{TermCache} \uplus \mathit{MixedCache} \\
\mathit{CallStack}^{\leq k} &:= (\mathcal{N}^{\neg\text{com}} \cdot \mathit{Cache})^{\leq k} \\
\mathit{DelayedControl} &:= \mathit{TermCache} \uplus \mathit{MixedCache} \uplus \mathit{NonTermCache} \\
\mathit{NormalControl} &:= (\mathcal{N} \cdot \mathit{Cache}) \uplus (\Sigma \cdot \mathcal{N} \cdot \mathit{Cache}) \uplus (\Sigma \cdot \mathit{Cache}) \\
\mathit{ControlState} &:= \mathit{NormalControl} \uplus \mathit{DelayedControl} \\
\gamma, \delta \in \mathit{Control}^{\leq k} &:= \mathit{ControlState} \cdot \mathit{CallStack}^{\leq k} \\
\mathit{Queue} &:= \mathbb{M}[\mathit{Msg}] \qquad & \mathit{Queues} &:= \mathit{Chan} \to \mathit{Queue} \\
\mathit{Config}^{\leq k} &:= \mathbb{M}\left[\mathit{Control}^{\leq k}\right] \times \mathit{Queues}
\end{aligned}
$$

Note that we assume the equality $\epsilon = \emptyset$ to simplify notation. We write $\mathit{Control}^{\mathbb{M}} := \mathit{Control}^{\leq \infty}$ and $\mathit{CallStack}^{\mathbb{M}} := \mathit{CallStack}^{\leq \infty}$. [3]

Definition 4 (Alternative Sequential Semantics). Let $\mathcal{G}$ be a PCCFG. We define a transition relation $\rightarrow_{\text{seq}'}$ on $\mathit{Control}^{\mathbb{M}}$ by induction over the following rules:

If $A \to B\,C$ is a $\mathcal{G}$-rule, C commutative and $C \rightarrow^*_{\text{seq}} w \in (\mathcal{N}^{\text{com}} \cup \Sigma^{\text{com}})^*$ then

$$A\,M\,\gamma \rightarrow_{\text{seq}'} B\,(\mathbb{M}(w) \,\oplus\, M)\,\gamma \tag{7}$$

[3] Defining Cache as a distjoint union enables a definition by cases according to the type of cache, thus rendering $\rightarrow_{\text{con}'}$ monotone with respect to an ordering.

If $A \to B\,C$ is a $\mathcal{G}$-rule and C non-commutative then

$$A\,M\,\gamma \to_{\text{seq}'} B\,C\,M\,\gamma \tag{8}$$

If $A \to a\,B$ is a $\mathcal{G}$-rule and $a \in \Sigma$ and $B \in \mathcal{N}$ then

$$A\,M\,\gamma \to_{\text{seq}'} a\,B\,M\,\gamma \tag{9}$$

If $A \to a$ is a $\mathcal{G}$-rule where $a \in \Sigma \cup \{\epsilon\}$ then

$$A\,M\,\gamma \to_{\text{seq}'} a\,M\,\gamma \tag{10}$$

where $\gamma \in \mathit{CallStack}^{\mathbb{M}}$, $M \in \mathit{Cache}$, and A, B and C range over non-terminals.

From the alternative sequential semantics, we derive a corresponding alternative concurrent semantics, using the following notation: for $M \in \mathbb{M}[\Sigma^{\text{com}}]$ and $w \in (\Sigma^{\text{com}})^*$

$$\begin{array}{ll} \Gamma \oplus \Gamma' := \{c \mapsto \Gamma(c) \oplus \Gamma'(c) \mid c \in \mathit{Chan}\} & \\ \Gamma(M) := \left\{c \mapsto \textstyle\sum_{c!m \in M} M(c\,!\,m) \mid c \in \mathit{Chan}\right\} & \Gamma(w) := \Gamma(\mathbb{M}(w)) \\ \Pi(M) := \{X \mapsto M(\nu X) \mid X \in \mathcal{N}\} & \Pi(w) := \Pi(\mathbb{M}(w)) \end{array}$$

Definition 5 (Alternative Concurrent Semantics). We define a binary relation $\to_{\text{con}'}$ over $\mathbb{M}[\mathit{Control}^{\mathbb{M}}] \times (\mathit{Chan} \to \mathbb{M}[\mathit{Msg}])$ by induction over the following rules:

If $\gamma \in \mathit{NormalControl} \cdot \mathit{CallStack}^{\mathbb{M}}$, $\gamma \to_{\text{seq}'} \gamma'$ then

$$\gamma \parallel \Pi \lhd \Gamma \to_{\text{con}'} \gamma' \parallel \Pi \lhd \Gamma \tag{11}$$

If $(c\,?\,m)\,\gamma \in \mathit{NormalControl} \cdot \mathit{CallStack}^{\mathbb{M}}$, $m \in \mathit{Msg}$ then

$$(c\,?\,m)\,\gamma \parallel \Pi \lhd ([m] \oplus q)^c, \Gamma \to_{\text{con}'} \gamma \parallel \Pi \lhd q^c, \Gamma \tag{12}$$

If $X \in \mathcal{N}$, $(\nu X)\,\gamma \in \mathit{NormalControl} \cdot \mathit{CallStack}^{\mathbb{M}}$ then

$$(\nu X)\,\gamma \parallel \Pi \lhd \Gamma \to_{\text{con}'} \gamma \parallel X \parallel \Pi \lhd \Gamma \tag{13}$$

If $(c\,!\,m)\,\gamma \in \mathit{NormalControl} \cdot \mathit{CallStack}^{\mathbb{M}}$, $m \in \mathit{Msg}$ then

$$(c\,!\,m)\,\gamma \parallel \Pi \lhd q^c, \Gamma \to_{\text{con}'} \gamma \parallel \Pi \lhd ([m] \oplus q)^c, \Gamma \tag{14}$$

If $l\,\gamma \in \mathit{NormalControl} \cdot \mathit{CallStack}^{\mathbb{M}}$, $l \in \mathcal{L}$ then

$$l\,\gamma \parallel \Pi \lhd \Gamma \to_{\text{con}'} \gamma \parallel \Pi \lhd \Gamma \tag{15}$$

If $M\,X\,\gamma \in \mathit{DelayedControl} \cdot \mathit{CallStack}^{\mathbb{M}}$, $M \in \mathit{TermCache}$, $\Gamma' = \Gamma \oplus \Gamma(M)$, $\Pi' = \Pi \oplus \Pi(M)$ then

$$M\,X\,\gamma \parallel \Pi \lhd \Gamma \to_{\text{con}'} X\,\gamma \parallel \Pi' \lhd \Gamma' \tag{16}$$

If $M\,\gamma \in \mathit{DelayedControl} \cdot \mathit{CallStack}^{\mathbb{M}}$, $M \in \mathit{MixedCache}$, $\Gamma' = \Gamma \oplus \Gamma(M)$, $\Pi' = \Pi \oplus \Pi(M)$ and $M' = M \upharpoonright (\mathcal{N}^{\text{com}} \cup \mathcal{L})$ then

$$M\,\gamma \parallel \Pi \lhd \Gamma \to_{\text{con}'} M'\,\gamma \parallel \Pi' \lhd \Gamma' \tag{17}$$

The alternative semantics precomputes the actions of commutative non-terminals on the call stacks. This is achieved by rule (7) in the alternative sequential semantics. The rules (16) and (17) are the concurrent counterparts; they ensure that the precomputed actions are rendered effective at the appropriate moment. Rule (16) is applicable when the precomputed cache M contains exclusively commutative actions; such a cache denotes a

sequence of commutative non-terminals whose computation terminates and generates concurrency actions. Rule (17), on the other hand, handles the case where the cache M contains non-terminals. An interpretation of such a cache is a partial computation of a sequence of commutative non-terminals. In this case rule (17) dispatches all commutative actions and then blocks. It is necessary to consider this case since not all non-terminals have terminating computations. Thus rule (7) may non-deterministically decide to abandon the pre-compution of actions.

We give a variant of the program-point coverability problem tailored to the alternative semantics and show its equivalence with the program-point coverabilily problem.

Verification Problem 2 (Alternative Program-Point Coverability). Given an APCPS $\mathcal{G}$ and a set of program point labels $l_1, \ldots, l_n$, a tuple $(P; l_1, ..., l_n)$ is a yes-instance of the *alternative program-point coverability* problem just if there exist a $\rightarrow_{\text{con}'}$-reachable configuration $\Pi \lhd \Gamma$ such that for every $i \in \{1, \ldots, n\}$ there exists $\lambda_i \, \gamma_i \in \Pi$ such that either $\lambda_i = l_i$, or $\lambda_i = M_i$ and $l_i \in M_i$?

In the long version of this paper [24] we show that the standard semantics *weakly simulates* the alternative semantics for APCPS. Thus for every configuration reachable in the alternative semantics there is a corresponding configuration reachable in the standard semantics. Owing to the nature of precomputations and caches, it is more difficult to relate runs of the standard semantics to those of the alternative semantics. However, for every run in the standard semantics reaching a configuration, there exists a run in the alternative semantics reaching a corresponding configuration.

Theorem 1 (Reduction of Program-Point Coverability). *A tuple $(P; l_1, \ldots, l_n)$ is a yes-instance of the program-point coverabililty problem if, and only if, $(P; l_1, \ldots, l_n)$ is a yes-instance of the alternative program-point coverability problem.*

4 APCPS with Shaped Stacks

In this section we present a natural restriction on the shape of the call stacks of APCPS processes. This shape restriction says that, at all times, at most an *a priori* fixed number of non-commutative non-terminals may reside in the call stack. Because the restriction does not apply to commutative non-terminals, call stacks can grow to arbitrary heights. We show that the alternative semantics for such shape-constrained APCPS gives rise to a well-structured transition system, thus allowing us to show the decidability of the alternative program-point coverability problem.

Definition 6. Define $Reach_{\rightarrow_{\text{con}'}} := \{\Pi \lhd \Gamma \mid [S] \lhd \emptyset \rightarrow^*_{\text{con}'} \Pi \lhd \Gamma\}$. Let $k \in \mathbb{N}$, we say an APCPS $\mathcal{G}$ *has k-shaped stacks* just if $Reach_{\rightarrow_{\text{con}'}} \subseteq Config^{\leq k}$. An APCPS $\mathcal{G}$ *has shaped stacks* just if $\mathcal{G}$ has k-shaped stacks for some $k \in \mathbb{N}$.

It follows from the definition that, in the alternative semantics, processes of an APCPS with k-shaped stacks have the form: $\gamma \, X_1 \, M_1 \, X_2 \, M_2 \cdots X_j \, M_j$ where $\gamma \in ControlState$, $X_i \in \mathcal{N}^{\neg\text{com}}$ and $j \leq k$. Relating this to the standard semantics, processes of an APCPS with k-shaped stacks are always of the form $\alpha \, X_1 \, \beta_1 \, X_2 \, \beta_2 \cdots X_j \, \beta_j$ where $\alpha \in (\mathcal{N} \cup (\Sigma \cdot \mathcal{N}) \cup (\Sigma \cup \{\epsilon\})) \cdot \mathcal{N}^{\text{com}*}$ and $\beta_i \in \mathcal{N}^{\text{com}*}$. It is this shape that lends itself to the name APCPS. Even though the shaped stacks constraint is semantic, we can give a *syntactic* sufficient condition: (the simple proof is omitted.)

Proposition 1. *Let $\mathcal{G}$ be an APCPS. If there is a well-founded partial order $\geq_{shape}$ such that for every $A \in \mathcal{N}$ and $B \in RHS(A) \cap \mathcal{N}$: (i) $A \geq_{shape} B$, and (ii) $\exists C \in \mathcal{N}^{\neg com}$: $A \to B\,C$ is a $\mathcal{G}$-rule $\Rightarrow$ $A >_{shape} B$, then $\mathcal{G}$ has shaped stacks.*

Example 3. Proposition 1 tells us that the program in Figures 1 and 2 can be modelled by an APCPS with shaped stacks. Non-tail recursive calls are potentially problematic. In our example the recursive call to setup_network() in the definition of setup_network is non-tail recursive, but only places a send action on the call stack, thus causing no harm. The only other non-tail recursive calls occur in do_task: the call to decompose_task poses no threat since decompose_task does not invoke do_task again. The two recursive calls to do_task either place procedure calls with send or no concurrent actions on the stack.

4.1 APCPS with Shaped Stacks and Well-Structured Transition Systems

We will now show the decidability of the alternative program-point coverability problem for APCPS with shaped stacks. First we recall the definition of well-structured transition systems [15]. Let $\leq$ be an ordering over a set U; we say $\leq$ is a *well-quasi-order* (wqo) just if for all infinite sequences $u_1, u_2, \ldots$ there exists i, j such that $u_i \leq u_j$. A *well-structured transition system* (WSTS) is a quadruple $(S, \to, \leq, s_0)$ such that $s_0 \in S$, $\leq$ is a wqo over S and $\to \subseteq S \times S$ is monotone with respect to $\leq$, i.e. if $s \to s'$ and $s \leq t$ then there exists t' such that $t \to t'$.

WSTS are an expressive class of infinite state systems that enjoy good model checking properties. A decision problem for WSTS of particular interest to verification is the *coverability problem* i.e. given a state s is it the case that $s_0 \to^* s'$ and $s \leq s'$. For $U \subseteq S$ define the sets $Pred(U) := \{s \mid s \to u, u \in U\}$ and $\uparrow U := \{u' \mid u \leq u', u \in U\}$. For WSTS the coverability problem is decidable [15] provided that for any given $s \in S$ the set $\uparrow Pred(\uparrow \{s\})$ is effectively computable. Wqos can be composed in various ways which makes decision results for WSTS applicable to a wide variety of infinite state models. In the following we recall a few results on the composition of wqos.

(WQO-a) If $(A_i, \leq_i)$ are wqo sets for $i = 1, ..., k$ then $(A_1 \times \cdots \times A_k, \leq_1 \times \cdots \times \leq_k)$ is a wqo set. (*Dickson's Lemma*)

(WQO-b) If A is a finite set then $(A, =)$ is a wqo set.

(WQO-c) If $(A, \leq)$ is a wqo then $(\mathbb{M}[A], \leq_{\mathbb{M}[A]})$ is a wqo set where $M_1 \leq_{\mathbb{M}[A]} M_2$ just if for all $a \in A$ there exists an $a' \geq a$ such that $M_1(a) \leq M_2(a')$ [33].

(WQO-d) If $(A, \leq_A)$ and $(B, \leq_B)$ are wqo sets, then $(A \cdot B, \leq_A \cdot \leq_B)$ is a wqo set, where $\gamma \cdot \gamma' \leq_A \cdot \leq_B \delta \cdot \delta$ just if $\gamma \leq_A \delta$ and $\gamma' \leq_B \delta'$.

(WQO-e) If $(A, \leq_A)$ and $(B, \leq_B)$ are wqo set, then $(A \uplus B, \leq_A \uplus \leq_B)$ is a wqo set, where $a \leq_A \uplus \leq_B b$ just if $a, b \in A$ and $a \leq_A b$ or $a, b \in B$ and $a \leq_B b$.

4.2 A Well-Quasi-Order for the Alternative Semantics

Fix a k. Our goal is to construct a well-quasi-order for $Config^{\leq k}$ as a first step to showing the alternative semantics gives rise to a WSTS for APCPS with shaped stacks.

We order the multi-sets $TermCache$, $NonTermCache$, $MixedCache$ and $Queue$ with the multi-set inclusion $\leq_{\mathbb{M}}$ which is a well-quasi-order. Since $Chan$ is a finite

set and $Queues = Chan \to \mathbb{M}[Msg] \cong \mathbb{M}[Msg]^{|Chan|}$ we obtain a well-quasi-order for $Chan \to \mathbb{M}[Msg]$ using a generalisation of Dickson's lemma. We then compose the wqo of $TermCache$ and $MixedCache$ to obtain a wqo $\leq_{Cache} := \leq_{TermCache} \uplus \leq_{MixedCache}$ for $Cache$. For each $j \in \{1 \dots k\}$ we define

$$X_1\, M_1\, X_2\, M_2 \cdots X_j\, M_j \leq X_1\, M_1'\, X_2\, M_2' \cdots X_j\, M_j' \quad \text{iff} \quad \forall i : M_i \leq_{Cache} M_i'$$

which gives a well-quasi-order for $CallStack^{\leq k}$. We obtain a wqo for $DelayedControl$ by composing the wqos of $TermCache$, $NonTermCache$ and $MixedCache$:

$$\leq_{DelayedControl} := \leq_{TermCache} \uplus \leq_{NonTermCache} \uplus \leq_{MixedCache} .$$

Since Σ and $\mathcal{N}$ are finite sets, $(\Sigma, =_\Sigma)$ and $(\mathcal{N}, =_\mathcal{N})$ are wqo sets, and so, we can compose a wqo for $NormalControl$:

$$\leq_{NormalControl} := (=_\Sigma \cdot \leq_{Cache}) \uplus (=_\Sigma \cdot =_\mathcal{N} \cdot \leq_{Cache}) \uplus (=_\mathcal{N} \cdot \leq_{Cache}) .$$

Similarly we can construct wqos for $ControlState$ and $Control^{\leq k}$ by composition:

$$\leq_{ControlState} := \leq_{NormalControl} \uplus \leq_{DelayedControl}$$
$$\leq_{Control^{\leq k}} := \leq_{ControlState} \cdot \leq_{CallStack^{\leq k}} .$$

As a last step we use (WQO-c) to construct a wqo for $\mathbb{M}\left[Control^{\leq k}\right]$ which then allows us to define a wqo for $Config^{\leq k}$ by $\leq_{Config^{\leq k}} := \leq_{\mathbb{M}\left[Control^{\leq k}\right]} \times \leq_{Queues}$.

To prove the decidability of the coverability problem for APCPS with shaped stacks, it remains to show that $\to_{\text{con}'}$ is monotonic and $\uparrow Pred(\uparrow \{\gamma\})$ is computable.

Lemma 1 (Monotonicity). *The transition relation $\to_{con'}$ is monotone with respect to the well-order $\leq_{Config^{\leq k}}$.*

Corollary 1. *The transition system $\left(Config^{\leq k}, \to_{con'}, \leq_{Config^{\leq k}}\right)$ is a well-structured transition system.*

To see that $\uparrow Pred(\uparrow \{\gamma\})$ is computable is mostly trivial; only predecessors generated by rule (7) are not immediately obvious. Given $M' \in Cache$ we observe that it is enough to be able to compute the set $P_{M'} := \uparrow \{(C, M) \mid C \in \mathcal{N}^{\text{com}}, C \to^*_{\text{seq}} w, M'' = M \oplus \mathbb{M}(w), M' \leq_{\mathbb{M}} M''\}$. Now $C \to^*_{\text{seq}} w$ is a computation of a commutative context-free grammar (CCFGs) for which an encoding into Petri nets has been shown by Ganty and Majumdar [17]. Their encoding builds on work by Esparza [11] modelling CCFG in Petri nets. Their translation leverages a recent result [14]: every word of a CCFG has a *bounded-index* derivation i.e. every term of the derivation uses no more than an *a priori* fixed number of occurrences of non-terminals. A budget counter constrains the Petri net encoding of a CCFG to respect boundedness of index; termination of a CCFG computation can be detected by a transition that is only enabled when the full budget is available. This result allows us to compute the set $P_{M'}$ using a backwards coverability algorithm for Petri nets.

Theorem 2. *The alternative program-point coverability problem, and hence the program-point coverability problem, for APCPS with k-shaped stacks are decidable for every $k \geq 0$.*

5 Related Work and Discussion

Partially Commutative Context-Free Grammars (PCCFG). Czerwinski et al. introduced PCCFG as a study in process algebra [7]. They proved that bisimulation is NP-complete for a class of processes extending BPA and BPP [11] where the sequential composition of certain processes is commutative. Bisimulation is defined on the traces of such processes, although there is no synchronisation between processes. In [8] the problem of word reachability for partially commutative context-free languages was shown to be NP-complete.

Asynchronous Procedure Calls. Petri net models for finite state machines that communicate asynchronously via unordered message buffers were first investigated by Mukund et al. [27, 28]. In an influential paper [30] in 2006, Sen and Viswanathan showed that safety verification is decidable for first-order programs with atomic asynchronous methods. Building on this, Jhala and Majumdar [22] constructed a VAS that models such asynchronous programs on-the-fly. Liveness properties, such as fair termination and starvation, of asynchronous programs were extensively studied by Ganty et al. in [18, 17]. In our more general APCPS framework, we may view the asynchronous programs considered by Ganty and Majumdar in [17] as APCPS running a single "scheduler" process. Task bags can be modelled as channels in our setting and the posting of a task can be modelled by sending a message; the scheduling of a procedure call can be simulated as a receive of a non-deterministically selected channel which unlocks a commutative procedure call defined by rules of types (i) and (ii) and rules of type (iii) where $C \in \mathcal{N}^{\mathrm{com}}$, in the sense of Definition 2. It is thus easy to see that APCPS with shaped stacks subsume programs with asynchronous procedure calls. In light of the fact that their safety verification is EXPSPACE-complete we can infer that the program-point coverability problem for APCPS with shaped stacks is EXPSPACE-hard.

Various extensions of Sen and Viswanathan's model [6] and applications to real-world asynchronous task scheduling systems [19] have been investigated. From the standpoint of message-passing concurrency, a key restriction of many of the models considered is that messages may only be retrieved by a communicating pushdown process when its stack is empty. The aim of this paper is to relax this restriction while retaining decidability of safety verification.

Communicating Pushdown Systems. The literature on communicating pushdown systems is vast. Numerous classes with decidable verification problems have been discovered. Heußner et al. [21] studied a restriction on pushdown processes that communicate asynchronously via FIFO channels: a process may send a message only when its stack is empty, while message retrieval is unconstrained. Several other communicating pushdown systems have been explored: parallel flow graph systems [13], visibly pushdown automata that communicate over FIFO-queues [1], pushdown systems communicating over locks [23], and recursive programs with hierarchical communication [4, 2].

Verification techniques that over-approximate correctness properties of concurrent pushdown systems have been studied [16, 20]. Under-approximation techniques typically impose constraints, such as bounding the number of context switches [32, 25], bounding the number of times a process can switch from a send-mode to receive-mode [3], or allowing symbols pushed onto the stack to be popped only within a bounded

number of context switches [31]. Another line of work focuses on pushdown systems that communicate synchronously over channels, restricting model checking to synchronisation traces that fall within a restricted regular language [12]; this approach has been developed into an effective CEGAR method [26].

Future Directions and Conclusion. We have introduced a new class of asynchronously communicating pushdown systems, APCPS, and shown that the program-point coverability problem is decidable and EXPSPACE-hard for the subclass of APCPS with shaped stacks. We plan to investigate the precise complexity of the program-point coverability problem, construct an implementation and integrate it into SOTER [9, 10], a safety verifier for Erlang programs, to study APCPS empirically.

Acknowledgments. Financial support by EPSRC (research grant EP/F036361/1 and OUCL DTG Account doctoral studentship for the first author) is gratefully acknowledged. We would like to thank Matthew Hague, Subodh Sharma, Michael Tautschnig and Emanuele D'Osualdo for helpful discussions and insightful comments, and the anonymous reviewers for their detailed reports.

References

[1] Babic, D., Rakamaric, Z.: Asynchronously communicating visibly pushdown systems. Technical Report UCB/EECS-2011-108, UC Berkeley (2011) 5
[2] Bouajjani, A., Emmi, M.: Analysis of recursively parallel programs. In: POPL, pp. 203–214 (2012) 5
[3] Bouajjani, A., Emmi, M.: Bounded phase analysis of message-passing programs. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 451–465. Springer, Heidelberg (2012) 5
[4] 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) 5
[5] Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM 30(2), 323–342 (1983) 1
[6] Chadha, R., Viswanathan, M.: Decidability results for well-structured transition systems with auxiliary storage. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 136–150. Springer, Heidelberg (2007) 5
[7] Czerwiński, W., Fröschle, S., Lasota, S.: Partially-commutative context-free processes. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 259–273. Springer, Heidelberg (2009) 1, 2, 5
[8] Czerwiński, W., Hofman, P., Lasota, S.: Reachability problem for weak multi-pushdown automata. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 53–68. Springer, Heidelberg (2012) 5
[9] D'Osualdo, E., Kochems, J., Ong, C.-H.L.: Soter: an automatic safety verifier for Erlang. In: AGERE! 2012, pp. 137–140 (2012) 5
[10] D'Osualdo, E., Kochems, J., Ong, C.-H.L.: Automatic verification of Erlang-style concurrency. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 454–476. Springer, Heidelberg (2013) 5
[11] Esparza, J.: Petri nets, commutative context-free grammars, and basic parallel processes. Fundam. Inform. 31(1), 13–25 (1997) 4.2, 5
[12] Esparza, J., Ganty, P.: Complexity of pattern-based verification for multithreaded programs. In: POPL, pp. 499–510 (2011) 2, 5

[13] Esparza, J., Podelski, A.: Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In: POPL, pp. 1–11 (2000) 5
[14] Esparza, J., Ganty, P., Kiefer, S., Luttenberger, M.: Parikh's theorem: A simple and direct construction. CoRR, abs/1006.3825 (2010) 2, 4.2
[15] Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science 256(1-2), 63–92 (2001) 4.1
[16] Flanagan, C., Qadeer, S.: Thread-modular model checking. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 213–224. Springer, Heidelberg (2003) 5
[17] Ganty, P., Majumdar, R.: Algorithmic verification of asynchronous programs. ACM Trans. Program. Lang. Syst. 34(1), 6 (2012) 1, 4.2, 5
[18] Ganty, P., Majumdar, R., Rybalchenko, A.: Verifying liveness for asynchronous programs. In: POPL, pp. 102–113 (2009) 5
[19] Geeraerts, G., Heußner, A., Raskin, J.-F.: Queue-dispatch asynchronous systems. CoRR, abs/1201.4871 (2012) 5
[20] Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread-modular abstraction refinement. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 262–274. Springer, Heidelberg (2003) 5
[21] Heußner, A., Leroux, J., Muscholl, A., Sutre, G.: Reachability analysis of communicating pushdown systems. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 267–281. Springer, Heidelberg (2010) 5
[22] Jhala, R., Majumdar, R.: Interprocedural analysis of asynchronous programs. In: POPL, pp. 339–350 (2007) 1, 5
[23] Kahlon, V.: Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise CFL-reachability for threads communicating via locks. In: LICS, pp. 27–36 (2009) 5
[24] Kochems, J., Ong, C.-H.L.: Safety verification of asynchronous pushdown systems with shaped stacks (long version) (2013), `http://www.cs.ox.ac.uk/people/jonathan.kochems/apcps.pdf` 1, 3
[25] Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. Formal Methods in System Design 35(1), 73–97 (2009) 5
[26] Long, Z., Calin, G., Majumdar, R., Meyer, R.: Language-theoretic abstraction refinement. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 362–376. Springer, Heidelberg (2012) 5
[27] Mukund, M., Narayan Kumar, K., Radhakrishnan, J., Sohoni, M.: Towards a characterisation of finite-state message-passing systems. In: Hsiang, J., Ohori, A. (eds.) ASIAN 1998. LNCS, vol. 1538, pp. 282–299. Springer, Heidelberg (1998) 5
[28] Mukund, M., Narayan Kumar, K., Radhakrishnan, J., Sohoni, M.: Robust asynchronous protocols are finite-state. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 188–199. Springer, Heidelberg (1998) 5
[29] Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. 22(2), 416–430 (2000) 1
[30] 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) 1, 5
[31] La Torre, S., Napoli, M.: Reachability of multistack pushdown systems with scope-bounded matching relations. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 203–218. Springer, Heidelberg (2011) 5
[32] 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) 5
[33] Wehrman, I.: Higman's theorem and the multiset order (2006), `http://www.cs.utexas.edu/~iwehrman/pub/ms-wqo.pdf` 4.1

Reversibility and Asymmetric Conflict in Event Structures

Iain Phillips[1] and Irek Ulidowski[2]

[1] Department of Computing, Imperial College London, England
[2] Department of Computer Science, University of Leicester, England

Abstract. Reversible computation has attracted increasing interest in recent years, with applications in hardware, software and biochemistry. We introduce reversible forms of prime event structures and asymmetric event structures. In order to control the manner in which events are reversed, we use asymmetric conflict on events. We discuss, with examples, reversing in causal order, where an event is only reversed once all events it caused have been reversed, as well as forms of non-causal reversing.

1 Introduction

Causal reversibility in concurrent systems means that events that cause other events can only be undone after the caused events are undone first, and that events which are independent of each other can be reversed in an arbitrary order. The last decade has produced a good understanding of how causal reversibility can be achieved in the settings of operational semantics and process calculi. Research on reversing process calculi can be traced back perhaps to Berry and Boudol's Chemical Abstract Machine [2]. Danos and Krivine reversed CCS [5,6], a general method for reversing process calculi was proposed in [13], and reversible structures that compute forwards and backwards asynchronously were developed by Cardelli and Laneve [4]. Mechanisms for controlling reversibility based on a rollback construct were devised by Lanese, Mezzina, Schmitt and Stefani [9] for a reversible higher-order π calculus [10], and an alternative mechanism based on the execution control operator was proposed in [15].

Perhaps with the exception of [15] and [16], other common forms of reversibility, such as *inverse causal* reversibility, have not been studied yet. In [16] we present an initial study of a form of reversible event structure based on a generalisation of Winskel's enabling relation [19]. In this paper we propose reversible event structures which are strongly contrasted to those of [16], as we here focus on analysing conflict and causation as first-class notions in the setting of reversible computation, rather than maximising expressive power.

We here take the view that reversing an event a means that a is removed from the current configuration (a set of events which have occurred and have not been reversed), and it is as if a had never occurred, apart possibly from indirect effects, such as a having caused another event b before a was reversed.

Our motivating example is the basic catalytic cycle for protein substrate phosphorylation by a kinase. We describe how bonds are created and dissolved in

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 303–318, 2013.

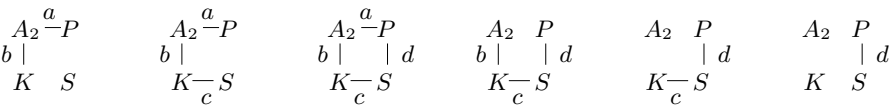


Fig. 1. Basic catalytic cycle for substrate phosphorylation by a kinase

the cycle as presented in [18, Figure 1a]. A kinase K aims to transfer a phosphate group P from a nucleotide Adenosine TriPhospate (ATP), which has three phosphate groups, to a protein substrate S. After the transfer ATP will become Adenosine DiPhosphate (ADP), and so we denote ATP as $A_2 - P$, where the bond between A_2 and P is a, and ADP as A_2. Firstly, $A_2 - P$ and then S bind to the active site of K. We denote the bonds thus created as b and c respectively; see Figure 1, which should be read from left to right. Then phosphorylation takes place: P is transferred from $A_2 - P$ to a Ser, Thr or Tyr residue of S by creating the bond d and then dissolving a. Finally A_2 and then S is released from the active site of K, so b and then c is broken. We note that the order in which bonds are created and broken differs for different kinases in such catalytic cycles [18]; hence we seek a general method for reversing events in an arbitrary order. Let events a, b, c, d represent the bonds a, b, c, d. The order in which bonds are created can be defined by the *causality* relation $<$ of *prime event structures* (PES) [12,19]: $a < b < c < d$. To express undoing of events we shall add to PES a new *reverse causality* relation $\prec$: here $a \prec \underline{a}$, $b \prec \underline{b}$ and $c \prec \underline{c}$ mean that a, b, c can be reversed (notation $\underline{a}, \underline{b}, \underline{c}$) as long as they have happened, and $d \prec \underline{a}$, $d \prec \underline{b}$, $d \prec \underline{c}$ force undoing of a, b, c only after d. We do not include $d \prec \underline{d}$, since d is irreversible here. We force that a is undone before b is undone by extending PES further with a *prevention* relation $\triangleright$: $a \triangleright \underline{b}$ prevents undoing of b while a is present; similarly $b \triangleright \underline{c}$. Thus, we obtain a *reversible* PES (RPES). The resulting forward transitions between configurations are $(\emptyset \rightarrow)\{a\} \rightarrow \{a, b\} \rightarrow \{a, b, c\} \rightarrow \{a, b, c, d\}$ and reverse transitions are $\{a, b, c, d\} \rightarrow \{b, c, d\} \rightarrow \{c, d\} \rightarrow \{d\}$. This is an example of *inverse* causal reversibility: a is reversed before undoing b even though a causes b, similarly for b and c. See [15,16] for other examples of non-causal reversibility.

There is a deficiency in the RPES solution in that, for example, a can occur again (so to speak) in configurations $\{b, c, d\}, \{c, d\}, \{d\}$. A general remedy is to add forwards prevention $e \triangleright e'$ to the reverse prevention $e \triangleright \underline{e'}$ already present in RPESs to obtain *reversible* asymmetric event structures (RAES). These are a reversible version of the *asymmetric event structures* (AES) of Baldan, Corradini and Montanari [1]. Prevention $e \triangleright e'$ is *asymmetric* conflict, where both e and e' can happen, but only if e' occurs before e. This generalises the symmetric conflict relation $e \sharp e'$ of PESs. If we add $d \triangleright a$ (d prevents a from taking place) to our example then this disallows a in $\{b, c, d\}, \{c, d\}$ and $\{d\}$.

There are two standard ways of explaining causation. Event a *causes* event b ($a < b$) means either (1) in any run (computation), if b occurs then a occurs earlier or (2) if b is enabled at configuration X then we must have $a \in X$. The two views are equivalent if there is no reversing. Suppose that we have three events

with $a < b < c$. On view (1) we deduce that $a < c$. On view (2) we also deduce that $a < c$, provided that X is left-closed (downwards closed under $<$), which will be the case for forward-only computation. Thus causation is transitive, as is the case in PESs and AESs.

In the context of reversible computation the second view of causation is simpler, and that is the one that we adopt. If all reversing is causal then all configurations will still be left-closed, and so it is still natural to require $<$ to be transitive. However once we admit the possibility of non-causal reversing, which leads to non-left-closed configurations (such as $\{b, c, d\}$ and $\{c, d\}$ in our example), it is no longer reasonable to insist on $<$ being transitive; if $a < b < c$ then a may have been reversed after b occurs, and before c occurs. Therefore, when defining RAESs we allow causation to be non-transitive. This extension is somewhat orthogonal to the move from symmetric to asymmetric conflict. We introduce the concept of *sustained causation*, where $a \ll b$ means that a causes b and a cannot reverse until b reverses. This is the analogue of standard causation for forwards computation, and we take sustained causation to be transitive.

We also consider the issue of conflict inheritance (if $a < b$ and $a \sharp c$ then $b \sharp c$) in the reversible setting. If $a < b$ and $a \sharp c$ and a is reversible, then we can undo a in $\{a, b\}$ to reach $\{b\}$. Now there is nothing in $\{b\}$ to prevent c from taking place, and so we expect that $\{b, c\}$ is a configuration, and b and c are not in conflict. Hence, there is no conflict inheritance with $<$. However, we still have conflict inheritance with respect to sustained causality $a \ll b$.

We assign meaning to the structures we consider by defining *configuration systems*, which are transition systems with configurations as states and sets of concurrent events as labels. It is natural to allow *mixed* transitions, which perform both forward and reverse moves. We are not aware of models with mixed transitions having been considered previously.

In this paper we present an account of conflict and causation in the reversible, and not necessarily causal, setting. We define RPESs and RAESs, and relate them to their respective forward-only counterparts. We prove a number of results about reachable configurations. We show under what conditions reachable configurations which are finite are reachable by purely finite means (Theorems 3.16 and 4.15). We show that under causal reversing any reachable configuration is forwards reachable (Theorem 3.22), and we propose conditions for configurations to be reachable under inverse causal reversing (Theorems 3.25 and 4.20). We define mappings between our event structures and show that they preserve configuration systems or reachable configurations (Theorem 5.1).

2 Configuration Systems

In this section we describe the model of concurrency we shall use for assigning meaning to the event structures considered in this paper. An event structure will be interpreted as a *configuration system*. Configuration systems are closely related to another model of concurrency, namely *configuration structures*, which have a notion of *configuration* and a notion of concurrent or *step* transition.

These were introduced by van Glabbeek and Goltz, and later generalised by van Glabbeek and Plotkin.

Let $\mathfrak{P}(E)$ denote the powerset of a set E.

Definition 2.1 ([8,7]). *A* configuration structure *is a pair $\mathcal{C} = (E, \mathsf{C})$ where E is a set of events and $\mathsf{C} \subseteq \mathfrak{P}(E)$. For $X, Y \in \mathsf{C}$, we let $X \to Y$ if $X \subseteq Y$ and for every Z, if $X \subseteq Z \subseteq Y$ then $Z \in \mathsf{C}$.*

The idea is that all the (possibly infinitely many) events in $Y \setminus X$ are independent, and so can happen as a single step. Instead of $X \to Y$, we can write $X \stackrel{A}{\to} Y$ where $A = Y \setminus X$. Note that if $Y = X \cup \{a\}$ and $X, Y \in \mathsf{C}$ then $X \to Y$. This may no longer hold in the reversible setting. As an example, let $E = \{a, b\}$. Suppose that a causes b, so that b cannot occur unless a has already occurred. Then $\{b\}$ is not a possible configuration using forwards computation. However if a is reversible, we can do a followed by b, followed by reversing a, and we reach $\{b\}$. Thus both $\emptyset$ and $\{b\}$ are configurations, but we do not have $\emptyset \stackrel{b}{\to} \{b\}$.

We propose a new definition appropriate for the reversible setting. We first establish our notation. We let $e, a, b, c, \ldots$ range over events, and $A, B, X, Y, Z, \ldots$ range over sets of events. If an event e is reversible, we have a corresponding reverse event $\underline{e}$. We write $\underline{B}$ for $\{\underline{e} : e \in B\}$. We let $\alpha, \ldots$ range over events or reverse events, and $\Delta, \ldots$ range over sets of events or reverse events.

Definition 2.2. *A* configuration system *is a quadruple $\mathcal{C} = (E, F, \mathsf{C}, \to)$ where E is a set of events, $F \subseteq E$ are the reversible events, $\mathsf{C} \subseteq \mathfrak{P}(E)$ is the set of* configurations *and $\to \subseteq \mathsf{C} \times \mathfrak{P}(E \cup \underline{F}) \times \mathsf{C}$ is a labelled transition relation such that if $X \stackrel{A \cup \underline{B}}{\to} Y$ then:*

- *$A \cap X = \emptyset$ and $B \subseteq X \cap F$ and $Y = (X \setminus B) \cup A$;*
- *for every $A' \subseteq A$ and $B' \subseteq B$ we have $X \stackrel{A' \cup \underline{B'}}{\to} Z \stackrel{(A \setminus A') \cup (\underline{B \setminus B'})}{\to} Y$ (where $Z = (X \setminus B') \cup A' \in \mathsf{C}$).*

We say that $A \cup \underline{B}$ is enabled *at X if there is Y such that $X \stackrel{A \cup \underline{B}}{\to} Y$. We say that a transition $X \stackrel{A \cup \underline{B}}{\to} Y$ is* mixed *if both A and B are non-empty. If $B = \emptyset$ we say the transition is* forwards*, and if $A = \emptyset$ the transition is* reverse*.*

The labels on the transitions are optional since they can be deduced from the configurations: if $X \stackrel{\Delta}{\to} Y$ then $\Delta = (Y \setminus X) \cup (\underline{X \setminus Y})$.

We define various kinds of configuration (cf. [8, Definition 3.5]):

Definition 2.3. *Let $\mathcal{C} = (E, F, \mathsf{C}, \to)$ be a configuration system and let $X \in \mathsf{C}$.*

- *X is a* forwards secured *configuration if there is an infinite sequence of configurations $X_i \in \mathsf{C}$ $(i = 0, \ldots)$ with $X = \bigcup_{i=0}^{\infty} X_i$ and $X_0 = \emptyset$ and $X_i \stackrel{A_{i+1}}{\to} X_{i+1}$ with $A_{i+1} \subseteq E$;*
- *X is a* reachable *configuration if there is some sequence $\emptyset \stackrel{A_1 \cup \underline{B}_1}{\to} \cdots \stackrel{A_n \cup \underline{B}_n}{\to} X$ where $A_i \subseteq E$ and $B_i \subseteq F$ for each $i = 1, \ldots, n$;*

- X *is a* forwards reachable *configuration if there is some sequence* $\emptyset \xrightarrow{A_1} \cdots \xrightarrow{A_n} X$ *where* $A_i \subseteq E$ *for each* $i = 1, \ldots, n$*;*
- X *is a* finitely reachable *configuration if there is some sequence* $\emptyset \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} X$ *where* $\alpha_i \in E \cup \underline{F}$ *for each* $i = 1, \ldots, n$*.*

Note that mixed transitions $X \xrightarrow{A \cup \underline{B}} Y$ do not yield new reachable sets compared to forward and reverse transitions, since if $X \xrightarrow{A \cup \underline{B}} Y$ then there is Z such that $X \xrightarrow{A} Z \xrightarrow{\underline{B}} Y$. However mixed transitions allow us to express the independence of forward and reverse events.

It is clear that the finitely reachable configurations are finite and are reachable configurations. However we shall see that it is not necessarily the case that finite, reachable configurations are finitely reachable (Example 3.15).

3 Reversing in Prime Event Structures

In this section we recall the definition of prime event structure, and formulate the slightly weaker notion of *pre-prime event structure*, which is more suitable for reversing events, since it does not require conflict to be hereditary. These pre-PESs will form the forward component of reversible PESs. We then introduce reversible prime event structures and study their properties.

We shall see that pre-PESs and PESs can be used interchangeably in forward-only computation, since they yield the same forwards secured configurations. On the other hand, when reverse computation is considered, then pre-PESs allow us to reach configurations that are not reachable with PESs.

Prime Event Structures. We start by recalling the definition of unlabelled prime event structures with binary conflict:

Definition 3.1 ([12]). *A* prime event structure *(PES) is a triple* $\mathcal{E} = (E, <, \sharp)$ *where* E *is a set of events,* $< \subseteq E \times E$ *is the causality relation, which is an irreflexive partial order such that for every* $e \in E$*,* $\{e' \in E : e' < e\}$ *is finite, and* $\sharp \subseteq E \times E$ *is the conflict relation, which is irreflexive, symmetric and* hereditary *with respect to* $<$*: if* $a < b$ *and* $a \sharp c$ *then* $b \sharp c$ *(all* $a, b, c \in E$*).*

When we generalise this definition to the reversible setting, we shall see that conflict heredity with respect to $<$ no longer necessarily holds. We therefore formulate a weaker form of prime event structure, as follows.

Definition 3.2. *A* pre-prime event structure *(pre-PES) is a triple* $\mathcal{E} = (E, <, \sharp)$ *where* E *is a set of events and*

1. $\sharp \subseteq E \times E$ *is irreflexive and symmetric;*
2. $< \subseteq E \times E$ *is an irreflexive partial order such that for every* $e \in E$*,* $\{e' \in E : e' < e\}$ *is finite and conflict-free;*
3. *if* $a < b$ *then not* $a \sharp b$ *(all* $a, b \in E$*).*

Here X *is* conflict-free *means that for all* $a, b \in X$*, it is not the case that* $a \sharp b$*.*

It is straightforward to check that any PES is also a pre-PES. Note that if X is conflict-free and $Y \subseteq X$ then Y is also conflict-free.

Definition 3.3. *Let $\mathcal{E} = (E, <, \sharp)$ be a pre-PES. We define the associated configuration system $C(\mathcal{E}) = (E, \emptyset, \mathsf{C}, \rightarrow)$ as follows. Let $\mathsf{C} = \{X \subseteq E : X$ is conflict-free$\}$. For $X \in \mathsf{C}$ and $A \subseteq E$, we say that A is* enabled *at X if $A \cap X = \emptyset$, $X \cup A$ is conflict-free, and for every $a \in A$, $\{b \in E : b < a\} \subseteq X$. We define $X \xrightarrow{A} Y$ iff $X, Y \in \mathsf{C}$ and $Y = X \cup A$ and A is enabled at X.*

It can be checked that if $\mathcal{E} = (E, <, \sharp)$ is a pre-PES then $C(\mathcal{E})$ satisfies the definition of a configuration system.

Definition 3.4. *Let $\mathcal{E} = (E, <, \sharp)$ be a pre-PES. We define the* causal depth *of an event $e \in E$ by $\mathsf{cdepth}(e) = \max\{\mathsf{cdepth}(e') + 1 : e' < e\}$, where we conventionally let $\max(\emptyset) = 0$.*

Causal depth is always finite, since each event has only finitely many causes. Events with no causes have depth zero.

Let $\mathcal{E} = (E, <, \sharp)$ be a pre-PES and let $X \subseteq E$. We say that X is *left-closed (under $<$)* if for any $a \in X$, if $b < a$ then $b \in X$.

Proposition 3.5. *Let $\mathcal{E} = (E, <, \sharp)$ be a pre-PES and let $C(\mathcal{E}) = (E, \emptyset, \mathsf{C}, \rightarrow)$.*

1. *The forwards secured configurations in C are precisely those which are left-closed.*
2. *$X \in \mathsf{C}$ is (forwards) reachable iff X is left-closed and there is $k \in \mathbb{N}$ such that for all $e \in X$, $\mathsf{cdepth}(e) < k$.*

Pre-PESs are no more expressive than PESs as far as configuration systems are concerned. Any pre-PES $\mathcal{E}$ can be converted into a corresponding PES $\mathsf{hc}(\mathcal{E})$ by taking the hereditary closure of the conflict relation, in such a way that the configuration systems have the same forwards secured configurations and the same transitions on the reachable portion:

Example 3.6. Let $\mathcal{E} = (E, <, \sharp)$ where $E = \{a, b, c\}$ and $a < b$, $a \sharp c$. Then $\mathcal{E}$ is a pre-PES with configurations $\emptyset, \{a\}, \{c\}, \{a, b\}, \{b, c\}$. The corresponding PES $\mathsf{hc}(\mathcal{E})$ is the same, except that we have $b \sharp c$ by conflict heredity, and therefore $\{b, c\}$ is not a configuration. However $\mathcal{E}$ and $\mathsf{hc}(\mathcal{E})$ have the same reachable configurations, as $\{b, c\}$ is not reachable in $\mathcal{E}$.

Note that in Example 3.6, if a were to become reversible we could reach $\{b, c\}$ in $\mathcal{E}$ (but not in $\mathsf{hc}(\mathcal{E})$) by performing $a, b, \underline{a}, c$, and the two structures would no longer be equivalent.

Reversible Prime Event Structures. We now introduce reversible PESs.

Definition 3.7. *A reversible prime event structure (RPES) is a sextuple $\mathcal{E} = (E, F, <, \sharp, \prec, \triangleright)$ where $(E, <, \sharp)$ is a pre-PES, $F \subseteq E$ are those events of E which are reversible, with reverse events being denoted by $\underline{F} = \{\underline{e} : e \in F\}$ and*

1. $\triangleright \subseteq E \times \underline{F}$ *is the* prevention *relation;*
2. $\prec \subseteq E \times \underline{F}$ *is the* reverse causality *relation, where we require* $a \prec \underline{a}$ *for each* $a \in F$*, and also that* $\{a : a \prec \underline{b}\}$ *is finite and conflict-free for every* $b \in F$*;*
3. *if* $a \prec \underline{b}$ *then not* $a \triangleright \underline{b}$*;*
4. $\sharp$ *is hereditary with respect to* sustained causation $\ll$*: if* $a \ll b$ *and* $a \sharp c$ *then* $b \sharp c$*, where we define* $a \ll b$ *to mean that* $a < b$ *and if* $a \in F$ *then* $b \triangleright \underline{a}$*;*
5. $\ll$ *is transitive.*

The intended meaning of $a \prec \underline{b}$ is that for b to be reversed, a must be present in the current configuration. This is a similar concept to forward causation. The intended meaning of $a \triangleright \underline{b}$ is that $\underline{b}$ cannot occur while a is in the current configuration. This has similarities with asymmetric conflict [11,17,1].

Note that $a \ll b$, which prevents a from being reversed once b has occurred (and until b is reversed), has something of the force of normal irreversible causation. Items (4) and (5) of Definition 3.7 could be replaced by stating that $(E, \ll, \sharp)$ is a PES.

Definition 3.8. *Let* $\mathcal{E} = (E, F, <, \sharp, \prec, \triangleright)$ *be an RPES. Let* $X \subseteq E$ *be conflict-free. For* $A \subseteq E$*,* $B \subseteq F$*, we say that* $A \cup \underline{B}$ *is* enabled *at* X *if*

- $A \cap X = \emptyset$*,* $B \subseteq X$ *and* $X \cup A$ *is conflict-free;*
- *for every* $a \in A$*, if* $c < a$ *then* $c \in X \setminus B$*;*
- *for every* $b \in B$*, if* $d \prec \underline{b}$ *then* $d \in X \setminus (B \setminus \{b\})$*;*
- *for every* $b \in B$*, if* $d \triangleright \underline{b}$ *then* $d \notin X \cup A$*.*

Definition 3.9. *Let* $\mathcal{E} = (E, F, <, \sharp, \prec, \triangleright)$ *be an RPES. We define the associated configuration system* $C(\mathcal{E}) = (E, F, \mathsf{C}, \rightarrow)$ *as follows. Let* $\mathsf{C} = \{X \subseteq E : X \text{ is conflict-free}\}$*. For* $X \in \mathsf{C}$ *and* $A \subseteq E$*,* $B \subseteq F$*, we define* $X \xrightarrow{A \cup \underline{B}} Y$ *iff* $X, Y \in \mathsf{C}$ *and* $Y = (X \setminus B) \cup A$ *and* $A \cup \underline{B}$ *is enabled at* X*.*

Proposition 3.10. *Let* $\mathcal{E}$ *be an RPES. Then* $C(\mathcal{E})$ *is a configuration system.*

Example 3.11. Consider $\mathcal{E}$ with $E = F = \{a, b, c\}$ and $a \ll b \ll c$ (where $\ll$ is sustained causation), and $a \prec \underline{a}$, $b \prec \underline{b}$ and $c \prec \underline{c}$. Note that we can deduce $a \ll c$ by transitivity of $\ll$. When we are in a configuration that contains b we cannot undo a, and we cannot undo a and b when c is present. All subsets of E are the configurations of $C(\mathcal{E})$; the reachable ones are $\emptyset, \{a\}, \{a, b\}, \{a, b, c\}$. On reachable configurations, the forwards transitions are $\emptyset \xrightarrow{a} \{a\} \xrightarrow{b} \{a, b\} \xrightarrow{c} \{a, b, c\}$ and the reverse transitions are $\{a, b, c\} \xrightarrow{c} \{a, b\} \xrightarrow{b} \{a\} \xrightarrow{a} \emptyset$. Hence, the events are reversed in causal order.

Reachable Configurations. We now explore how adding reversibility changes what configurations are reachable. Sustained causation $\ll$ in the reversible setting behaves somewhat like standard causation $<$ in the forwards-only setting.

Proposition 3.12. *Let* $\mathcal{E}$ *be an RPES and* $C(\mathcal{E}) = (E, F, \mathsf{C}, \rightarrow)$*.*

1. *The forwards secured configurations in* C *are precisely those which are left-closed under* $<$.
2. $X \in \mathsf{C}$ *is forwards reachable iff* X *is left-closed under* $<$ *and there is* $k \in \mathbb{N}$ *such that for all* $e \in X$, $\mathsf{cdepth}(e) < k$.
3. *If* $X \in \mathsf{C}$ *is reachable then* X *is left-closed under* $\ll$ *and there is* $k \in \mathbb{N}$ *such that for all* $e \in X$, $\mathsf{cdepth}(e) < k$.

The converse to Proposition 3.12(3) is false in view of the following example.

Example 3.13. Let $\mathcal{E} = (E, F, <, \sharp, \prec, \triangleright)$ be given by $E = \{a, b, c\}$ and $F = \{a\}$, with $a < b < c$ and $a \prec \underline{a}$, $c \prec \underline{a}$ (and empty $\sharp$ and $\triangleright$). Then $\{b\}$ is not a reachable configuration, although it is left-closed under $\ll$.

It is sometimes useful to see the reverse causation relation $\prec$ as between pairs of events, rather than events and reverse events, as this reveals chains of causality.

Definition 3.14. *Let* $\mathcal{E} = (E, F, <, \sharp, \prec, \triangleright)$ *be an RPES. For* $a \in E, b \in F$, *let* $a \prec^\bullet b$ *iff* $a \prec \underline{b}$ *and* $a \neq b$. *We also write* $b \succ^\bullet a$ *iff* $a \prec^\bullet b$.

The next example shows that, unlike in the forwards-only setting, we can have reachable configurations which are finite but not finitely reachable.

Example 3.15. Let $E = F = \{a_i : i \in \mathbb{N}\}$. Suppose also that $a_i \prec \underline{a}_i$, $a_{2i+1} < a_{2i}$ and $a_{2i+2} \prec \underline{a}_{2i+1}$ $(i \in \mathbb{N})$. There is no $\sharp$ or $\triangleright$. Then $\mathcal{E} = (E, F, <, \emptyset, \prec, \emptyset)$ is an RPES. The configuration $\{a_0\}$ is reachable in four steps as follows:

$$\emptyset \xrightarrow{\{a_1, a_3, \ldots\}} \{a_1, a_3, \ldots\} \xrightarrow{\{a_0, a_2, \ldots\}} E \xrightarrow{\{a_1, a_3, \ldots\}} \{a_0, a_2, \ldots\} \xrightarrow{\{a_2, a_4, \ldots\}} \{a_0\}$$

However with single-event transitions we can reach $\{a_0, a_{2i+1}\}$ for any $i \in \mathbb{N}$, but not $\{a_0\}$. Note that there is an infinite descending causal chain $a_0 > a_1 \succ^\bullet a_2 > a_3 \succ^\bullet \ldots$.

To ensure that every finite, reachable configuration is finitely reachable, we need to impose an extra condition on RPESs.

Theorem 3.16. *Let* $\mathcal{E} = (E, F, <, \sharp, \prec, \triangleright)$ *be an RPES. Suppose that for every* $e \in E$, $\{e' \in E : e'(< \cup \prec^\bullet)^* e\}$ *is finite. Then every finite, reachable configuration in* $C(\mathcal{E})$ *is finitely reachable.*

Proposition 3.17. *Let* $\mathcal{E}$ *be a RPES such that* $C(\mathcal{E})$ *has a reachable configuration* X *with* $X \xrightarrow{b}$ *(some* $b \in F$*). Then* $C(\mathcal{E})$ *has a non-terminating computation.*

Reversing Disciplines. Many patterns of common biochemical reactions involve breaking of previously established bonds out-of-causal order [18]. We now consider several particular disciplines for reversing events out of many possible disciplines. The most usual is where we require that an event cannot be reversed until all events it has caused have also been reversed; we call this *cause-respecting*. A stronger notion is *causal*, where in addition to cause-respecting a reversible event can be reversed freely if all events it has caused have been reversed.

Definition 3.18. *Let $\mathcal{E} = (E, F, <, \sharp, \prec, \triangleright)$ be an RPES. We say that $\mathcal{E}$ is* cause-respecting *if for any $a, b \in E$, if $a < b$ then $a \ll b$, so that all causation is sustained causation. We say that $\mathcal{E}$ is* causal *if for any $a \in E$, $b \in F$, we have (1) $a \prec \underline{b}$ iff $a = b$ and (2) $a \triangleright \underline{b}$ iff $b < a$.*

We have already seen a causal RPES in Example 3.11. Any PES can be converted into a causal RPES, once we decide which events are to be reversible.

Proposition 3.19. *Let $\mathcal{E} = (E, <, \sharp)$ be a PES and let $F \subseteq E$. Define $\kappa(\mathcal{E}, F) = (E, F, <, \sharp, \prec, \triangleright)$, where $a \prec \underline{a}$ (all $a \in F$) and $a \triangleright \underline{b}$ for every $a \in E$, $b \in F$ such that $b < a$. Then $\kappa(\mathcal{E}, F)$ is a causal RPES.*

Next we investigate reachable configurations in cause-respecting RPESs.

Proposition 3.20. *Let $\mathcal{E}$ be a cause-respecting RPES and $C(\mathcal{E}) = (E, F, \mathsf{C}, \rightarrow)$. If $X \in \mathsf{C}$ is reachable then X is left-closed.*

If an RPES is causal then any mixed transition can be inverted on left-closed configurations, provided that the events in the transition are reversible.

Proposition 3.21. *Let $\mathcal{E}$ be an RPES and let $C(\mathcal{E}) = (E, F, \mathsf{C}, \rightarrow)$. Let $X \in \mathsf{C}$ be left-closed and let $A, B \subseteq F$. Then: 1. If $\mathcal{E}$ is cause-respecting and $X \stackrel{B}{\Rightarrow} X'$ then $X' \stackrel{B}{\rightarrow} X$. 2. If $\mathcal{E}$ is causal and $X \stackrel{A \cup \underline{B}}{\rightarrow} X'$ then $X' \stackrel{B \cup \underline{A}}{\rightarrow} X$.*

The second statement of Proposition 3.21 is related to the Loop Lemma for RCCS [5, Lemma 6], which states that every forward transition has a corresponding reverse transition, and conversely.

Theorem 3.22. *Let $\mathcal{E}$ be a cause-respecting RPES and let $C(\mathcal{E}) = (E, F, \mathsf{C}, \rightarrow)$. If $X \in \mathsf{C}$ is reachable then X is forwards reachable.*

Theorem 3.22 is related to a result of Danos and Krivine for RCCS [5, Cor. 1].

We now consider a second reversing discipline.

Definition 3.23. *Let $\mathcal{E} = (E, F, <, \sharp, \prec, \triangleright)$ be an RPES. We say that $\mathcal{E}$ is* inverse cause-respecting *if for any $a \in E$, $b \in F$, if $a < b$ then $a \triangleright \underline{b}$. We say that $\mathcal{E}$ is* inverse causal *if for any $a \in E$, $b \in F$, we have (1) $a \prec \underline{b}$ iff $a = b$ and (2) $a \triangleright \underline{b}$ iff $a < b$.*

This allows reversing to start at any time with a $<$-minimal element of a configuration belonging to F. Plainly we can reach new configurations which are not forwards reachable.

Example 3.24. Let $\mathcal{E} = (E, F, <, \sharp, \prec, \triangleright)$ be an inverse causal RPES with $E = F = \{a, b, c\}$ and $a < b < c$ and no conflict. We also have $a \prec \underline{a}$, $b \prec \underline{b}$ and $c \prec \underline{c}$, and $a \triangleright \underline{b}$, $a \triangleright \underline{c}$ and $b \triangleright \underline{c}$ since $\mathcal{E}$ is inverse causal. The forwards reachable configurations are $\emptyset$, $\{a\}$, $\{a, b\}$ and $\{a, b, c\}$. Reversing from $\{a, b\}$ we can reach $\{b\}$. Reversing from $\{a, b, c\}$ we can reach $\{b, c\}$ followed by $\{c\}$, from which we can reach $\{a, c\}$. Thus every subset of E is a reachable configuration. The empty configuration is reachable from all configurations.

Theorem 3.25. *Let $\mathcal{E}$ be an inverse causal RPES with all events reversible and let $C(\mathcal{E}) = (E, E, \mathsf{C}, \rightarrow)$. Let $X \in \mathsf{C}$ be such that there is a finite bound k such that for any $e \in X$, $\mathsf{cdepth}(e) < k$. Then X is reachable.*

4 Reversing in Asymmetric Event Structures

In this section we increase the expressiveness of RPESs by modifying conflict to be asymmetric rather than symmetric, and also dropping the requirement that causation is transitive. The extra expressive power allows us to model more faithfully different forms of reversing events as exemplified in the Introduction.

Asymmetric Event Structures. We recall the definition of asymmetric event structures:

Definition 4.1 ([1, Definition 2.4]). *An* asymmetric event structure *(AES) is a triple $\mathcal{E} = (E, <, \lhd)$ where E is a set of events and for all $a, b \in E$*

1. *$\lhd \subseteq E \times E$ is the precedence relation (where we write $b \rhd a$ iff $a \lhd b$);*
2. *$< \subseteq E \times E$ is the causality relation, which is an irreflexive partial order, such that $\{e \in E : e < a\}$ is finite and $\lhd$ is acyclic on $\{e \in E : e \leq a\}$;*
3. *if $a < b$ then $a \lhd b$;*
4. *if $a \sharp c$ and $a < b$ then $b \sharp c$, where $\sharp$ is defined to be $\lhd \cap \rhd$.*

What we write as $a \lhd b$ was $a \nearrow b$ in [1]. The precedence relation has a dual interpretation. Thus $a \lhd b$ says that event a *weakly causes*, or *precedes* event b, meaning that if both a and b occur then a occurred first. Dually, $b \rhd a$ says that b *prevents* a, meaning that if b is present in a configuration then a cannot occur. We have already used prevention $b \rhd \underline{a}$ on reverse events with RPESs.

Definition 4.2. *Let $\mathcal{E} = (E, <, \lhd)$ be an AES. We define the associated configuration system $C(\mathcal{E}) = (E, \emptyset, \mathsf{C}, \rightarrow)$ as follows. Let C consist of those $X \subseteq E$ such that $\lhd$ is well-founded on X. For $X \in \mathsf{C}$ and $A \subseteq E$, we say that A is* enabled *at X if $A \cap X = \emptyset$, and for every $a \in A$, both $\{b \in E : b < a\} \subseteq X$ and $\{b \in E : b \rhd a\} \cap (X \cup A) = \emptyset$. We define $X \stackrel{A}{\rightarrow} Y$ iff $X, Y \in \mathsf{C}$ and $Y = X \cup A$ and A is enabled at X.*

It can be checked that if $\mathcal{E}$ is an AES then $C(\mathcal{E})$ is a configuration system. Moving from symmetric to asymmetric conflict increases expressive power:

Example 4.3. Consider the AES $\mathcal{E} = (E, <, \lhd)$ with $E = \{a, b\}$ and $a \lhd b$. Then $C(\mathcal{E})$ consists of all subsets of E and we have $\emptyset \stackrel{a}{\rightarrow} \{a\} \stackrel{b}{\rightarrow} \{a, b\}$ and $\emptyset \stackrel{b}{\rightarrow} \{b\}$. There is no pre-PES for this configuration system.

Definition 4.4. *Let $\mathcal{E} = (E, <, \lhd)$ be an AES with $C(\mathcal{E}) = (E, \emptyset, \mathsf{C}, \rightarrow)$. Let $X \in \mathsf{C}$. We define the* precedence depth *of events in X by a mapping from X to the ordinals given by $\mathsf{pdepth}_X(e) = \sup\{\mathsf{pdepth}_X(e') + 1 : e' \in X \text{ and } e' \lhd e\}$.*

Note that $\mathsf{pdepth}_X(e)$ will be a (not necessarily finite) ordinal number by well-foundedness of $\lhd$ on X in C.

Proposition 4.5. *Let $\mathcal{E}$ be an AES with $C(\mathcal{E}) = (E, \emptyset, \mathsf{C}, \rightarrow)$. Let $X \in \mathsf{C}$. Then X is a forwards secured configuration iff X is left-closed and for all $e \in X$, $\mathsf{pdepth}_X(e)$ is finite.*

Reversible Asymmetric Event Structures. We now introduce the generalisation of RPESs to the setting of asymmetric conflict and not necessarily transitive causation.

Definition 4.6. *A reversible asymmetric event structure (RAES) is a quadruple* $\mathcal{E} = (E, F, \prec, \lhd)$ *where* E *is a set of events and* $F \subseteq E$ *are those events of* E *which are reversible, and for any* $a, b, c, e \in E$ *and* $\alpha \in E \cup \underline{F}$*:*

1. $\lhd \subseteq (E \cup \underline{F}) \times E$ *is the precedence relation (with* $a \lhd b$ *iff* $b \rhd a$*), which is irreflexive;*
2. $\prec \subseteq E \times (E \cup \underline{F})$ *is the direct causation relation, which is irreflexive and well-founded, and such that* $\{e \in E : e \prec \alpha\}$ *is finite and* $\lhd$ *is acyclic on* $\{e \in E : e \prec \alpha\}$*;*
3. $a \prec \underline{a}$ *for all* $a \in F$*;*
4. *if* $a \prec \alpha$ *then not* $a \rhd \alpha$*;*
5. $a \prec\!\!\prec b$ *implies* $a \lhd b$*, where* sustained direct causation $a \prec\!\!\prec b$ *means that* $a \prec b$ *and if* $a \in F$ *then* $b \rhd \underline{a}$*;*
6. $\prec\!\!\prec$ *is transitive;*
7. *if* $a \sharp c$ *and* $a \prec\!\!\prec b$ *then* $b \sharp c$*, where* $\sharp$ *is defined to be* $\lhd \cap \rhd$*.*

We have combined the forwards causation $<$ of (R)PESs and reverse causation $\prec$ of RPESs into a single direct causation relation $\prec$; similarly we have combined the forwards precedence $\lhd$ of AESs and the reverse prevention $\rhd$ of RPESs into a single precedence relation $\lhd$. We remark that direct (or immediate) causation $\prec$ was used in flow event structures [3] (with symmetric conflict $\sharp$).

If we set $F = \emptyset$ in Definition 4.6 we get an AES, since all causation is sustained causation. However if $F \neq \emptyset$ then the forwards-only part of an RAES is a *proto-AES*, which is like an AES, except that causation is not transitive and conflict is not hereditary. We discussed the reasons for this in the Introduction. We also drop the requirement that if $a < b$ then $a \lhd b$ (though that appears in its sustained causation form in item 5 of Definition 4.6). This does not hold in general in the reversible context. Let $E = \{a, b\}$ and $F = \{a\}$, with $a \prec b$ and $a \prec \underline{a}$. Then we can perform $a, b, \underline{a}$ to reach $\{b\}$. At this point a is enabled. Thus it is not the case that $a \lhd b$, since that means a is disabled when b is present.

Definition 4.7. *Let* $\mathcal{E} = (E, F, \prec, \lhd)$ *be an RAES. Let* $X \subseteq E$ *be such that* $\lhd$ *is well-founded on* X*. For* $A \subseteq E$*,* $B \subseteq F$*, we say that* $A \cup \underline{B}$ *is* enabled *at* X *if*

- $A \cap X = \emptyset$*,* $B \subseteq X$*;*
- *for every* $a \in A$*, if* $c \prec a$ *then* $c \in X \setminus B$*;*
- *for every* $a \in A$*, if* $c \rhd a$ *then* $c \notin X \cup A$*;*
- *for every* $b \in B$*, if* $d \prec \underline{b}$ *then* $d \in X \setminus (B \setminus \{b\})$*;*
- *for every* $b \in B$*, if* $d \rhd \underline{b}$ *then* $d \notin X \cup A$*.*

Definition 4.8. *Let* $\mathcal{E} = (E, F, \prec, \lhd)$ *be an RAES. We define the associated configuration system* $C(\mathcal{E}) = (E, F, \mathsf{C}, \rightarrow)$ *as follows. Let* C *consist of those* $X \subseteq E$ *such that* $\lhd$ *is well-founded on* X*. For* $X \in \mathsf{C}$ *and* $A \subseteq E$*,* $B \subseteq F$*, we define* $X \stackrel{A \cup \underline{B}}{\rightarrow} Y$ *iff* $X, Y \in \mathsf{C}$ *and* $Y = (X \setminus B) \cup A$ *and* $A \cup \underline{B}$ *is enabled at* X*.*

If $\mathcal{E}$ is an RAES then $C(\mathcal{E})$ is a configuration system. We now give examples involving asymmetric conflict and non-transitive causation.

Example 4.9. We illustrate how asymmetric conflict can be used to control reversing. Let $\mathcal{E} = (E, F, \prec, \lhd)$ be defined as follows. Let $E = \{a_1, \ldots, a_n\}$ and $F = \{a_1, \ldots, a_{n-1}\}$. We have $a_i \prec a_{i+1}$ and $a_i \prec \underline{a}_i$ $(1 \leq i \leq n-1)$; also $a_i \rhd \underline{a}_{i+1}$ $(1 \leq i \leq n-2)$. So far $\mathcal{E}$ is inverse causal (Definition 3.23), and events which have already been reversed can re-occur. We now add asymmetric conflict $a_i \lhd a_j$ $(1 \leq i < j \leq n)$, which prevents such re-occurrences, and also $a_{i+1} \prec \underline{a}_i$ $(1 \leq i \leq n-1)$, which ensures that we make progress towards the goal of the final configuration $\{a_n\}$. Non-empty reachable configurations of $\mathcal{E}$ are of the form $\{a_i, a_{i+1}, \ldots, a_j\}$ $(1 \leq i \leq j \leq n)$. At $\{a_i, \ldots, a_j\}$ we see that a_{j+1} is enabled if $j < n$ and $\underline{a}_i$ is enabled if $i < j$; in fact the mixed $\{a_{j+1}, \underline{a}_i\}$ is concurrently enabled if $i < j < n$. Thus we have a kind of FIFO queue which must be non-empty (apart from the initial empty configuration). All computations terminate, showing that Proposition 3.17 does not apply to RAESs.

Example 4.10. Let $\mathcal{E} = (E, \prec, \lhd)$ with $E = \{a, b, c, d\}$ and $a \prec b \prec c$, $d \prec c$ and $a \lhd d \lhd a$. Also let $F = \{a\}$ and $a \prec \underline{a}$. Then $\mathcal{E} = (E, F, \prec, \lhd)$ is an RAES. Note that a and d are in conflict ($a \sharp d$) and they are both (direct or indirect) causes of c. The configuration system $C(\mathcal{E})$ has $\emptyset \xrightarrow{a} \{a\} \xrightarrow{b} \{a, b\} \xrightarrow{\underline{a}} \{b\} \xrightarrow{d} \{b, d\} \xrightarrow{c} \{b, c, d\}$, $\{a\} \xrightarrow{\underline{a}} \emptyset$ and $\emptyset \xrightarrow{d} \{d\}$, together with various unreachable configurations. So the example illustrates how in the reversible setting an event can have conflicting indirect causes and still occur.

Remark 4.11. The forward-only part of $\mathcal{E}$ in Example 4.10 forms a proto-AES $\mathcal{E}'$, and we can use it as an example of how a proto-AES can be converted into an AES. In $\mathcal{E}'$ of course c cannot ever occur. To get a corresponding AES $(E', <, \lhd')$, we must eliminate c, as it has conflicting causes. This gives $E' = \{a, b, d\}$. We then let $a < b$ and $a \lhd' b$ (in a more elaborate example we would have to take the transitive closure of $\prec$). Finally we set $a \lhd' d \lhd' a$, $b \lhd' d \lhd' b$ so that conflict is inherited. This gives an AES $\mathsf{htc}(\mathcal{E}')$. Its configuration system has the same forward secured configurations as $C(\mathcal{E}')$, with some unreachable configurations eliminated.

Reachable Configurations. As with RPESs, sustained causation in the reversible setting behaves like standard causation in the forwards-only setting. The next result is the analogue of Proposition 3.12 for RPESs.

Definition 4.12. *Let $\mathcal{E} = (E, <, \lhd)$ be an RAES with $C(\mathcal{E}) = (E, \emptyset, \mathsf{C}, \rightarrow)$. Let $X \in \mathsf{C}$, and suppose that $\lhd \cup \prec$ is well-founded on X. We define the* precedence causal depth *of events in X by a mapping from X to the ordinals given by* $\mathsf{pcdepth}_X(e) = \sup\{\mathsf{pcdepth}_X(e') + 1 : e' \in X \text{ and } e' \lhd e \text{ or } e' \prec e\}$.

Proposition 4.13. *Let $\mathcal{E}$ be an RAES, $C(\mathcal{E}) = (E, F, \mathsf{C}, \rightarrow)$ and $X \in \mathsf{C}$. Then:*

1. *X is a forwards secured configuration iff X is left-closed, $\lhd \cup \prec$ is well-founded on X and for all $e \in X$, $\mathsf{pcdepth}_X(e)$ is finite.*

2. *X is forwards reachable iff X is left-closed, $\lhd \cup \prec$ is well-founded on X and there is $k \in \mathbb{N}$ such that for all $e \in X$, $\mathsf{pcdepth}_X(e) < k$.*
3. *If X is reachable then X is left-closed under $\prec\!\!\prec$ and there is $k \in \mathbb{N}$ such that for all $e \in X$, $\mathsf{cdepth}(e) < k$.*

It is not necessarily the case that $\lhd \cup \prec$ is well-founded on reachable configurations, as the next example shows.

Example 4.14. Let $E = \{a, b, c\}$, $F = \{a\}$ with $a \prec b \prec c \lhd a$ and $a \prec \underline{a}$. Then $(E, F, \prec, \lhd)$ is an RAES. By Proposition 4.13 we know that $\{a, b, c\}$ is not forwards reachable, since it contains a $\lhd \cup \prec$-cycle. However it is reachable by the computation $\emptyset \xrightarrow{a} \xrightarrow{b} \{a, b\} \xrightarrow{a} \{b\} \xrightarrow{c} \xrightarrow{a} \{a, b, c\}$.

As in the case of RPESs, we can have reachable configurations which are finite but not finitely reachable; the RPES $\mathcal{E} = (E, F, <, \emptyset, \prec, \emptyset)$ of Example 3.15 is easily converted into an RAES $\mathcal{E}' = (E, F, < \cup \prec, \emptyset)$ with an empty precedence relation. As with RPESs, to ensure that every finite, reachable configuration is finitely reachable, we shall need to impose extra conditions on RAESs.

Definition 3.14 for $\prec^\bullet$ carries over to RAESs and the next result is the analogue of Theorem 3.16.

Theorem 4.15. *Let $\mathcal{E} = (E, F, \prec, \lhd)$ be an RAES. Suppose that for every $e \in E$, $\{e' \in E : e'(\prec \cup \prec^\bullet)^* e\}$ is finite. Then every finite, reachable configuration in $C(\mathcal{E})$ is finitely reachable.*

Reversing Disciplines. We can define what it means for an RAES to be cause-respecting or causal by a straightforward adaptation of Definition 3.18. As with RPESs, causal implies cause-respecting. Also as with RPESs (Theorem 3.22), in a cause-respecting RAES, we can show that reachable configurations are forwards reachable.

We would like to prove a version of Proposition 3.21, which states that if an RPES is causal then any mixed transition can be inverted on left-closed configurations, provided that the events of the transition are reversible. However that no longer holds in the setting of RAESs, as the next example shows.

Example 4.16. Let $E = F = \{a, b\}$ and let $a \lhd b$, $a \prec \underline{a}$ and $b \prec \underline{b}$. Then $\mathcal{E} = (E, F, \prec, \rhd)$ is a causal RAES. All configurations are forwards reachable and left-closed. Note that a cannot occur after b going forwards, but we can reverse a and b in either order. In particular, we have $\{a, b\} \xrightarrow{a} \{b\}$ but not $\{b\} \xrightarrow{a} \{a, b\}$.

Thus we need a different notion than causal (or cause-respecting).

Definition 4.17. *Let $\mathcal{E} = (E, F, \prec, \rhd)$ be an RAES. We say that $\mathcal{E}$ is* precedence-respecting *if for any $a \in F$, $b \in E$, if $a \lhd b$ then $b \rhd \underline{a}$. We say that $\mathcal{E}$ is* precedence/cause-respecting *if $\mathcal{E}$ is cause-respecting and precedence-respecting. We say that $\mathcal{E}$ is* precedence causal *if for any $a \in E$, $b \in F$, both (1) $a \prec \underline{b}$ iff $a = b$ and (2) $a \rhd \underline{b}$ iff $b \prec a$ or $b \lhd a$.*

Clearly, if $\mathcal{E}$ is precedence causal then $\mathcal{E}$ is precedence/cause-respecting.

We can now obtain the analogue of Proposition 3.21 for RPESs.

Proposition 4.18. *Let $\mathcal{E}$ be an RAES and let $C(\mathcal{E}) = (E, F, \mathsf{C}, \rightarrow)$. Let $X \in \mathsf{C}$ be left-closed and let $A, B \subseteq F$. Then: 1. If $\mathcal{E}$ is precedence/cause-respecting and $X \overset{B}{\rightarrow} X'$ then $X' \overset{B}{\rightarrow} X$. 2. If $\mathcal{E}$ is precedence causal and $X \overset{A \cup \underline{B}}{\rightarrow} X'$ then $X' \overset{B \cup \underline{A}}{\rightarrow} X$.*

Any AES can be converted into a precedence causal RAES, once we decide which events are to be reversible. (cf. Proposition 3.19). Finally, we can also adapt inverse causal reversing (Definition 3.23) to RAESs.

Definition 4.19. *Let $\mathcal{E} = (E, F, <, \sharp, \prec, \triangleright)$ be an RPES. We say that $\mathcal{E}$ is* inverse precedence causal *if for any $a \in E$, $b \in F$, both (1) $a \prec \underline{b}$ iff $a = b$ and (2) $a \triangleright \underline{b}$ iff $a \prec b$ or $a \lhd b$.*

We showed in Theorem 3.25 that in an inverse causal RPES with all events reversible we can reach all configurations with bounded causal depth.

Theorem 4.20. *Let $\mathcal{E}$ be an inverse precedence causal RAES with all events reversible and let $C(\mathcal{E}) = (E, E, \mathsf{C}, \rightarrow)$. Let $X \in \mathsf{C}$ be such that there is a forwards reachable $X' \in \mathsf{C}$ with $X \subseteq X'$. Then X is reachable.*

5 Mappings

In this section we show how our event structures are related.

The mappings hc and htc that we have introduced informally in Example 3.6 and Remark 4.11 show that pre-PESs and proto-AESs are no more expressive than PESs and AESs as far as configuration systems are concerned. Hence, here we only consider the four main event structures and the mappings between them as in Figure 2. For $\mathcal{E} = (E, <, \sharp)$ a PES, we define $\rho_{\mathrm{p}}(\mathcal{E}) = (E, \emptyset, <, \sharp, \emptyset, \emptyset)$ and $\alpha(\mathcal{E}) = (E, <, \lhd)$ where $\lhd = < \cup \sharp$. For $\mathcal{E} = (E, <, \lhd)$ an AES, we define $\rho_{\mathrm{a}}(\mathcal{E}) = (E, \emptyset, <, \lhd)$. For $\mathcal{E} = (E, F, <, \sharp, \prec_{\mathsf{r}}, \triangleright_{\mathsf{r}})$ an RPES, we define $\alpha_{\mathrm{r}}(\mathcal{E}) = (E, F, \prec, \lhd)$ where $\prec = < \cup \prec_{\mathsf{r}}$ and $\lhd = \ll \cup \sharp \cup \lhd_{\mathsf{r}}$. The next theorem shows that the mappings preserve configuration systems or reachable configurations.

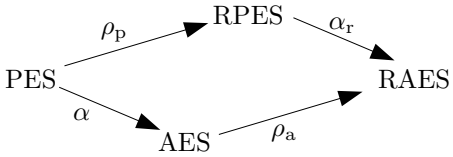


Fig. 2. Mappings

Theorem 5.1. *1. If $\mathcal{E}$ is a PES then $\rho_{\mathrm{p}}(\mathcal{E})$ is an RPES and $C(\rho_{\mathrm{p}}(\mathcal{E})) = C(\mathcal{E})$. Also, $\alpha(\mathcal{E})$ is an AES [1, Lemma 2.2] and X is reachable in $C(\mathcal{E})$ iff X is reachable in $C(\alpha(\mathcal{E}))$. 2. If $\mathcal{E}$ is an AES then $\rho_{\mathrm{a}}(\mathcal{E})$ is an RAES. Moreover, $C(\rho_{\mathrm{a}}(\mathcal{E})) = C(\mathcal{E})$. 3. If $\mathcal{E}$ is an RPES then $\alpha_{\mathrm{r}}(\mathcal{E})$ is an RAES. Moreover, X is reachable in $C(\mathcal{E})$ iff X is reachable in $C(\alpha_{\mathrm{r}}(\mathcal{E}))$.*

We now have two methods of mapping a PES into an RAES—via an AES or via an RPES— if $\mathcal{E}$ is a PES then $\alpha_{\mathrm{r}}(\rho_{\mathrm{p}}(\mathcal{E})) = \rho_{\mathrm{a}}(\alpha(\mathcal{E}))$.

6 Conclusions and Further Work

We have investigated conflict and causation for event structures with reversibility. We started by proposing a reversible form of prime event structure (RPES) where conflict inheritance no longer holds in general. The need for greater expressiveness then led us to two extensions: permitting non-transitive causation, and allowing asymmetric rather than symmetric conflict (useful for controlled reversing, as distinct from processes computing freely either forwards or backwards). These extensions yield our more general model, reversible asynchronous event structures (RAES). The two extensions are somewhat orthogonal and so one could envisage intermediate models. We have obtained results about which configurations are reachable. For instance we have given conditions under which finite and reachable configurations are guaranteed to be reachable without intermediate infinite configurations. Our models are general enough to allow several forms of reversibility to be defined and analysed, including the causal and inverse causal disciplines. We believe that RAESs offer the prospect of modelling a wide range of examples in software and biochemistry.

Future work could include formulating labelled versions of reversible event structures and bisimulations for them as in [14], establishing that our RPESs and RAESs are special cases of the reversible event structures in [16], modelling reversible process calculi, and extending existing work on domains and categories for event structures to the present models.

Acknowledgements. We thank the referees for their useful comments and suggestions. The second author thanks the University of Leicester for granting Academic Study Leave, acknowledges partial support from the JSPS Invitation Fellowship grant S13054, and thanks Shoji Yuen of Nagoya University.

References

1. Baldan, P., Corradini, A., Montanari, U.: Contextual Petri nets, asymmetric event structures, and processes. Information and Computation 171(1), 1–49 (2001)
2. Berry, G., Boudol, G.: The chemical abstract machine. Theoretical Computer Science 96(1), 217–248 (1992)
3. Boudol, G., Castellani, I.: Permutation of transitions: An event structure semantics for CCS and SCCS. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. LNCS, vol. 354, pp. 411–427. Springer, Heidelberg (1989)
4. Cardelli, L., Laneve, C.: Reversible structures. In: Proceedings of the 9th International Conference on Computational Methods in Systems Biology, pp. 131–140. ACM (2011)
5. Danos, V., Krivine, J.: Reversible communicating systems. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004)
6. 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)

7. van Glabbeek, R.J., Goltz, U.: Refinement of actions and equivalence notions for concurrent systems. Acta Informatica 37(4/5), 229–327 (2001)
8. van Glabbeek, R.J., Plotkin, G.D.: Configuration structures, event structures and Petri nets. Theoretical Computer Science 410(41), 4111–4159 (2009)
9. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.-B.: Controlling reversibility in higher-order pi. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 297–311. Springer, Heidelberg (2011)
10. 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)
11. Langerak, R.: Transformations and Semantics for LOTOS. PhD thesis, University of Twente (1992)
12. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, part I. Theoretical Computer Science 13, 85–108 (1981)
13. Phillips, I.C.C., Ulidowski, I.: Reversing algebraic process calculi. Journal of Logic and Algebraic Programming 73(1-2), 70–96 (2007)
14. Phillips, I.C.C., Ulidowski, I.: A hierarchy of reverse bisimulations on stable configuration structures. Mathematical Structures in Computer Science 22, 333–372 (2012)
15. Phillips, I.C.C., Ulidowski, I., Yuen, S.: A reversible process calculus and the modelling of the ERK signalling pathway. In: Glück, R., Yokoyama, T. (eds.) RC 2012. LNCS, vol. 7581, pp. 218–232. Springer, Heidelberg (2013)
16. Phillips, I.C.C., Ulidowski, I., Yuen, S.: Modelling of bonding with processes and events. In: Dueck, G.W., Miller, D.M. (eds.) RC 2013. LNCS, vol. 7948, pp. 141–154. Springer, Heidelberg (2013)
17. Pinna, G.M., Poigné, A.: On the nature of events. In: Havel, I.M., Koubek, V. (eds.) MFCS 1992. LNCS, vol. 629, pp. 430–441. Springer, Heidelberg (1992)
18. Ubersax, J.A., Ferrell, J.E.: Mechanisms of specificity in protein phosphorylation. Nature Reviews Molecular Cell Biology 8, 530–541 (2007)
19. Winskel, G.: Event structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 325–392. Springer, Heidelberg (1987)

The Power of Priority Channel Systems*

Christoph Haase, Sylvain Schmitz, and Philippe Schnoebelen

LSV, ENS Cachan & CNRS, France

Abstract. We introduce Priority Channel Systems, a new natural class of channel systems where messages carry a numeric priority and where higher-priority messages can supersede lower-priority messages preceding them in the fifo communication buffers. The decidability of safety and inevitability properties is shown via the introduction of a *priority embedding*, a well-quasi-ordering that has not previously been used in well-structured systems. We then show how Priority Channel Systems can compute Fast-Growing functions and prove that the aforementioned verification problems are $\mathbf{F}_{\varepsilon_0}$-complete.

1 Introduction

Channel systems are a family of distributed models where concurrent agents communicate via (usually unbounded) fifo communication buffers, called "channels". These models are well-suited for the formal specification and algorithmic analysis of communication protocols and concurrent programs [6, 7, 9]. They are also a fundamental model of computation, closely related to Post's tag systems.

A particularly interesting class of channel systems are the so-called *lossy channel systems (LCSs)*, where channels are unreliable and may lose messages [10, 4, 8]. For LCSs, several important behavioral properties, like safety or inevitability, are decidable. This is because these systems are *well-structured*: transitions are monotonic wrt. a (decidable) well-quasi-ordering of the configuration space [2, 14]. Beyond their applications in verification, LCSs have turned out to be an important automata-theoretic tool for decidability or hardness in areas like Timed Automata, Metric Temporal Logic, modal logics, etc. [3, 18, 23, 19]. They are also a fundamental model of computation capturing the $\mathbf{F}_{\omega^\omega}$-complexity level in Wainer *et al.*'s Fast-Growing Hierarchy, see [11, 24, 25].

Lossy channel systems do not provide a natural way to model systems or protocols that treat messages discriminatingly according to some specified rule set. An example is the prioritization of messages, which is central to ensuring *quality of service* (QoS) in networking architectures, and is usually implemented by allowing for tagging messages with some relative priority. For instance, the Differentiated Services (DiffServ) architecture described in RFC 2475, which enables QoS on modern IP networks, allows for a field specifying the relative priority of an IP packet with respect to a finite set of priorities, and network links may decide to arbitrarily drop IP packets of lower priority in favor of higher priority packets once the network congestion reaches a critical point.

* Work supported by the ReacHard project, ANR grant 11-BS02-001-01.

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 319–333, 2013.

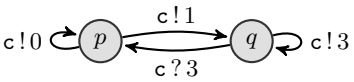


Fig. 1. A simple single-channel 3-PCS

Our contributions. We introduce *priority channel systems (PCSs)*, a family of channel systems where each message is equipped with a priority level, and where higher-priority messages can supersede lower-priority messages by dropping them. For technical simplicity, our model abstracts from the contents of messages by just considering the priority levels (but see the full version of this work at `http://arxiv.org/abs/1301.5500` for a general setting allowing arbitrary message contents and priorities).

Priority channel systems rely on the *(prioritized) superseding ordering*, a new ordering that has not been considered before in well-structured systems (though it is related to the gap-embedding of [28]). Showing that it is a well-quasi-ordering entails the decidability of safety and termination (among others) for PCSs. We also show the aforementioned problems to become undecidable for channel systems that build upon more restrictive priority mechanisms.

Using techniques from [24, 28], we show in Sec. 5 an $\mathbf{F}_{\varepsilon_0}$ upper bound on the complexity of PCS verification, far higher than the $\mathbf{F}_{\omega^\omega}$-complete complexity of LCSs. We then prove in Sec. 6 a matching lower bound and this is the main technical result of the paper: building upon techniques developed for less powerful models [11, 27, 17], we show how PCSs can robustly simulate the computation of the fast growing functions F_α and their inverses for all ordinals α up to ε_0. This gives a precise measure of the expressive power of PCSs.

Along the way, we show how other well-quasi-ordered data structures from the literature, e.g. trees with (strong) embedding, can be reflected in the superseding ordering (Sec. 4). This paves the way to new $\mathbf{F}_{\varepsilon_0}$ upper bounds for problems in other areas of algorithmic verification, whose complexity is wide open.

2 Priority Channel Systems

We define Priority Channel Systems as consisting of a single process since this is sufficient for our purposes in this paper.[1] For every $d \in \mathbb{N}$, the *level-d priority alphabet* is $\Sigma_d \stackrel{\text{def}}{=} \{0, 1, \ldots, d\}$. A *level-d priority channel system* (a "d-PCS") is a tuple $S = (\Sigma_d, \mathtt{Ch}, Q, \Delta)$ where Σ_d is as above, $\mathtt{Ch} = \{\mathtt{c}_1, \ldots, \mathtt{c}_m\}$ is a set of m *channel names*, $Q = \{q_1, q_2, \ldots\}$ is a finite set of *control states*, and $\Delta \subseteq Q \times \mathtt{Ch} \times \{!, ?\} \times \Sigma_d \times Q$ is a set of *transition rules* (see below).

2.1 Semantics

The operational semantics of a PCS S is given in the form of a transition system. We let $Conf_S \stackrel{\text{def}}{=} Q \times (\Sigma_d^*)^m$ be the set of all configurations of S, denoted $C, D, \ldots$

[1] Systems made of several concurrent components can be represented by a single process obtained as an asynchronous product of the components.

A configuration $C = (q, x_1, \ldots, x_m)$ records an instantaneous control point (a state in Q) and the contents of the m channels, i.e., sequences of messages from Σ_d. A sequence $x \in \Sigma_d^*$ has the form $x = a_1 \ldots a_\ell$ and we let $\ell = |x|$. Concatenation is denoted multiplicatively, with ε denoting the empty sequence.

The labeled transition relation between configurations, denoted $C \xrightarrow{\delta} C'$, is generated by the rules in $\Delta = \{\delta_1, \ldots, \delta_k\}$. From a technical perspective, it is convenient to define two such transition relations, denoted $\rightarrow_{\mathrm{rel}}$ and $\rightarrow_{\#}$.

Reliable Semantics. We start with $\rightarrow_{\mathrm{rel}}$ that corresponds to "reliable" steps, or more correctly steps with no superseding of lower-priority messages. As is standard, for a *reading rule* of the form $\delta = (q, \mathsf{c}_i, ?, a, q') \in \Delta$, there is a step $C \xrightarrow{\delta}_{\mathrm{rel}} C'$ if $C = (q, x_1, \ldots, x_m)$ and $C' = (q', y_1, \ldots, y_m)$ for some $x_1, y_1, \ldots, x_m, y_m$ such that $x_i = ay_i$ and $x_j = y_j$ for all $j \neq i$, while for a *writing rule* $\delta = (q, \mathsf{c}_i, !, a, q') \in \Delta$, there is a step $C \xrightarrow{\delta}_{\mathrm{rel}} C'$ if $y_i = x_i a$ (and $x_j = y_j$ for all $j \neq i$). These "reliable" steps correspond to the behavior of queue automata, or (reliable) channel systems, a Turing-powerful computation model.

Internal-Superseding. The actual behavior of PCSs is obtained by extending reliable steps with *internal superseding steps*, denoted $C \xrightarrow{\mathsf{c}_i \# k}_{\#} C'$, which can be performed at any time in an uncontrolled manner.

Formally, for two words $x, y \in \Sigma_d^*$ and $k \in \mathbb{N}$, we write $x \xrightarrow{\#k}_{\#} y \overset{\text{def}}{\Leftrightarrow} x$ is some $a_1 \ldots a_\ell$, $1 \leq k < |x| = \ell$, $a_k \leq a_{k+1}$ and $y = a_1 \ldots a_{k-1} a_{k+1} \ldots a_\ell$. In other words, the k-th message in x is superseded by its immediate successor a_{k+1}, with the condition that a_k is not of higher priority. We write $x \rightarrow_{\#} y$ when $x \xrightarrow{\#k}_{\#} y$ for some k, and use $x \leftarrow_{\#} y$ when $y \rightarrow_{\#} x$. The transitive reflexive closure $\overset{*}{\leftarrow}_{\#}$ is called the *superseding* ordering and is denoted by $\leq_{\#}$. Put differently, $\rightarrow_{\#}$ is a rewrite relation over Σ_d^* according to the rules $\{aa' \rightarrow a' \mid 0 \leq a \leq a' \leq d\}$.

This is extended to steps between configurations by $C = (q, x_1, \ldots, x_m) \xrightarrow{\mathsf{c}_i \# k}_{\#} C' = (q', y_1, \ldots, y_m) \overset{\text{def}}{\Leftrightarrow} q = q'$ and $x_i \xrightarrow{\#k}_{\#} y_i$ (and $x_j = y_j$ for $j \neq i$). Furthermore, every reliable step is a valid step: for any rule δ, $C \xrightarrow{\delta}_{\#} C'$ iff $C \xrightarrow{\delta}_{\mathrm{rel}} C'$, giving rise to a second transition system associated with S: $\mathcal{S}_{\#} \overset{\text{def}}{=} (Conf_S, \rightarrow_{\#})$. E.g., the PCS from Fig. 1 can perform

$$p, 0200 \xrightarrow{!1}_{\#} q, 02\underline{0}01 \xrightarrow{\#3}_{\#} q, \underline{0}201 \xrightarrow{\#1}_{\#} q, 2\underline{0}1 \xrightarrow{\#2}_{\#} q, 21 \,.$$

The internal-superseding semantics allows superseding to occur at any time and anywhere in the channel. It is appropriate when abstracting from situations where end-to-end communication actually goes through a series of consecutive relays, network switches, and buffers, each of them possibly handling the incoming traffic with a so-called *write-superseding* policy, where writes immediately "consume" the congested messages in front of them in the buffer. We develop this aspect in the full version, where we also prove the two semantics to be essentially equivalent.

2.2 Related Models

It is possible to consider a stricter policy for priorities where a higher-priority message may only supersede messages with *strictly* lower priority. Another priority mechanism one could envision sees higher-priority messages *overtake* those of lower priority without dropping them. These two mechanisms are more restrictive, i.e., drop fewer messages, but they may be powerless in case of network congestion: for instance, they offer no solutions if all the messages in the congested buffers have the same priority. From a more theoretical standpoint, both semantics also yield Turing-powerful models; details are provided in the full version.

Theorem 1. *Reachability in PCSs is undecidable both for strict superseding and overtaking semantics.*

We conclude by observing that PCSs can simulate lossy channel systems. In fact they can simulate the dynamic lossy channel systems and the timed lossy channel systems from [1], see the full version. Hence reachability and termination (see Thm. 3) are at least $\mathbf{F}_{\omega^\omega}$-hard for PCSs, and problems like boundedness or repeated control-state reachability (see [26] for more) are undecidable for them.

2.3 Priority Channel Systems Are Well-Structured

Our main result regarding the verification of PCSs is that they are *well-structured* systems. Recall that $C \leq_\# D \overset{\text{def}}{\Leftrightarrow} C$ is some $(p, y_1, \ldots, y_m)$ and D is $(p, x_1, \ldots, x_m)$ with $x_i \leq_\# y_i$ for $i = 1, \ldots, m$, or equivalently, C can be obtained from D by internal superseding steps.

Theorem 2 (PCSs are WSTSs). *For any PCS S, the transition system $\mathcal{S}_\#$ with configurations ordered by $\leq_\#$ is a well-structured transition system (with stuttering compatibility).*

Proof. There are two conditions to check:

1. **wqo:** $(Conf_S, \leq_\#)$ is a well-quasi-ordering as will be shown next (see Thm. 7 in Sec. 3).
2. **monotonicity:** Checking stuttering compatibility (see [14, def. 4.4]) is trivial with the $\leq_\#$ ordering. Indeed, assume that $C \leq_\# D$ and that $C \to_\# C'$ is a step from the "smaller" configuration. Then in particular $D \xrightarrow{*}_\# C$ by definition of $\to_\#$, so that clearly $D \xrightarrow{+}_\# C'$ and D can simulate any step from C. □

A consequence of the well-structuredness of PCSs is the decidability of several natural verification problems. In this paper we focus on "Reachability"[2] (given a PCS, an initial configuration C_0, and a set of configurations $G \subseteq Conf_S$, does $C_0 \xrightarrow{*}_\# D$ for some $D \in G$?), and "Inevitability" (do all maximal runs from C_0 eventually visit G?) which includes "Termination" as a special case.

[2] Also called "Safety" when we want to check that G is *not* reachable.

Theorem 3. *Reachability and Inevitability are decidable for PCSs.*

Proof (Sketch). The generic WSTS algorithms [14] apply after we check the minimal effectiveness requirements: the ordering $\leq_\#$ between configurations is decidable (in NLogSpace, see Sec. 3.2) and the operational semantics is finitely branching and effective (one can compute the immediate successors of a configuration, and the minimal immediate predecessors of an upward-closed set).

We note that Reachability and Coverability coincide (even for zero-length runs when C_0 has empty channels) since $\xrightarrow{+}_\#$ coincides with $\geq_\# \circ \xrightarrow{+}_\#$, and that the answer to a Reachability question only depends on the (finitely many) minimal elements of G. One can even compute $Pre^*(G)$ for G given, e.g., as a regular subset of $Conf_S$.

For Inevitability, the algorithms in [2, 14] assume that G is downward-closed but, in our case where $\xrightarrow{+}_\#$ and $\geq_\# \circ \xrightarrow{+}_\#$ coincide, decidability can be shown for arbitrary (recursive) G, as in [26, Thm. 4.4]. □

3 Priority Embedding

This section focuses on the superseding ordering $\leq_\#$ on words and establishes the fundamental properties we use for reasoning about PCSs. Recall that $\leq_\# \stackrel{\text{def}}{=} \xleftarrow{*}_\#$, the reflexive transitive closure of the inverse of $\rightarrow_\#$; we prove that $(\Sigma_p^*, \leq_\#)$ is a *well-quasi-ordering* (a wqo). Recall that a quasi-ordering $(X, \preccurlyeq)$ is a wqo if any infinite sequence $x_0, x_1, x_2, \ldots$ over X contains an infinite increasing subsequence $x_{i_0} \preccurlyeq x_{i_1} \preccurlyeq x_{i_2} \preccurlyeq \cdots$

3.1 Embedding with Priorities

For two words $x, y \in \Sigma_d^*$, we let $x \sqsubseteq_\mathrm{p} y \stackrel{\text{def}}{\Leftrightarrow} x = a_1 \cdots a_\ell$ and y can be factored as $y = z_1 a_1 z_2 a_2 \cdots z_\ell a_\ell$ with $z_i \in \Sigma_{a_i}^*$ for $i = 1, \ldots, \ell$. For example, $201 \sqsubseteq_\mathrm{p} 22011$ but $120 \not\sqsubseteq_\mathrm{p} 10210$ (factoring 10210 as $z_1 1 z_2 2 z_3 0$ needs $z_3 = 1 \notin \Sigma_0^*$). If $x \sqsubseteq_\mathrm{p} y$ then x is a subword of y and x can be obtained from y by removing factors of messages with priority not above the first preserved message to the right of the factor. In particular, $x \sqsubseteq_\mathrm{p} y$ implies $y \xrightarrow{*}_\# x$, i.e., $x \leq_\# y$. This immediately yields:

$$\varepsilon \sqsubseteq_\mathrm{p} y \text{ iff } y = \varepsilon\,, \tag{1}$$

$$x_1 \sqsubseteq_\mathrm{p} y_1 \text{ and } x_2 \sqsubseteq_\mathrm{p} y_2 \text{ imply } x_1 x_2 \sqsubseteq_\mathrm{p} y_1 y_2\,, \tag{2}$$

$$x_1 x_2 \sqsubseteq_\mathrm{p} y \text{ imply } \exists y_1 \sqsupseteq_\mathrm{p} x_1 : \exists y_2 \sqsupseteq_\mathrm{p} x_2 : y = y_1 y_2\,. \tag{3}$$

Lemma 4. *$(\Sigma_d^*, \sqsubseteq_p)$ is a quasi-ordering (i.e., is reflexive and transitive).*

Proof. Reflexivity is obvious from the definition. For transitivity, consider $x' \sqsubseteq_\mathrm{p} x \sqsubseteq_\mathrm{p} y$ with $x = a_1 \cdots a_\ell$ and $y = z_1 a_1 \cdots z_\ell a_\ell$. In view of Eqs. (1–3) it is enough to show $x' \sqsubseteq_\mathrm{p} y$ in the case where $|x'| = 1$. Consider then $x' = a$. Now $x' \sqsubseteq_\mathrm{p} x$ implies $a = a_\ell$ and $a \geq a_i$, hence $\Sigma_{a_i}^* \subseteq \Sigma_a^*$, for all $i = 1, \ldots, \ell$. Letting $z \stackrel{\text{def}}{=} z_1 a_1 \cdots z_{\ell-1} a_{\ell-1} z_\ell$ yields $y = za$ for $z \in \Sigma_a^*$. Hence $x' \sqsubseteq_\mathrm{p} z$. □

We can now relate superseding and priority orderings with:

Proposition 5. *For all $x, y \in \Sigma_d^*$, $x \sqsubseteq_p y$ iff $x \leq_\# y$.*

Proof. Obviously, $y \xrightarrow{\#k}_\# x$ allows $x \sqsubseteq_\mathrm{p} y$ with z_k being the superseded message (and $z_i = \varepsilon$ for $i \neq k$), so that $\leq_\#$ is included in $\sqsubseteq_\mathrm{p}$ by Lem. 4. In the other direction $x \sqsubseteq_\mathrm{p} y$ entails $x \leq_\# y$ as noted earlier. □

3.2 Canonical Factorizations and Well-quasi-ordering

For our next development, we define the *height*, written $h(x)$, of a sequence $x \in \Sigma_d^*$ as being the highest priority occurring in x (by convention, we let $h(\varepsilon) \stackrel{\text{def}}{=} -1$). Thus, $x \in \Sigma_h^*$ iff $h \geq h(x)$. (We further let $\Sigma_{-1} \stackrel{\text{def}}{=} \emptyset$.) Any $x \in \Sigma_d^*$ has a unique *canonical* factorization $x = x_0 h x_1 h \cdots x_{k-1} h x_k$ where k is the number of occurrences of $h = h(x)$ in x and where the $k+1$ *residuals* x_0, $x_1, \ldots, x_k$ are in Σ_{h-1}^*. The point of this decomposition is the following sufficient condition for $x \sqsubseteq_\mathrm{p} y$.

Lemma 6. *Let $x = x_0 h \cdots h x_k$ and $y = y_0 h \cdots h y_m$ be canonical factorizations with $h = h(x) = h(y)$. If there is a sequence $0 = j_0 < j_1 < j_2 < \cdots < j_{k-1} < j_k = m$ of indexes s.t. $x_i \sqsubseteq_p y_{j_i}$ for all $i = 0, \ldots, k$ then $x \sqsubseteq_p y$.*

Proof. We show $x \leq_\# y$. Note that $h y_i h \xrightarrow{*}_\# h$ for all $i = 1, \ldots, m$, so $y \xrightarrow{*}_\# y' \stackrel{\text{def}}{=} y_{j_0} h y_{j_1} h y_{j_2} \cdots h y_{j_k}$ (recall that $0 = j_0$ and $m = j_k$). From $x_i \sqsubseteq_\mathrm{p} y_{j_i}$ we deduce $y_{j_i} \xrightarrow{*}_\# x_i$ for all $i = 0, \ldots, k$, hence $y' \xrightarrow{*}_\# x_0 h \cdots h x_k = x$. □

The condition in the statement of Lemma 6 is usually written $\langle x_0, \ldots, x_k \rangle \preceq_* \langle y_0, \ldots, y_m \rangle$, using the *sequence extension* of $\sqsubseteq_\mathrm{p}$ on sequences of residuals.

Theorem 7. *$(\Sigma_d^*, \sqsubseteq_p)$ is a well-quasi-ordering (a wqo).*

Proof. By induction on d. The base case $d = -1$ is trivial since Σ_{-1}^* is $\emptyset^* = \{\varepsilon\}$, a singleton. For the induction step, consider an infinite sequence $x_0, x_1, \ldots$ over Σ_d^*. We can extract an infinite subsequence, where all x_i's have the same height h (since $h(x_i)$ is in a finite set) and, since the residuals are in Σ_{d-1}^*, a wqo by ind. hyp., further extract an infinite subsequence where the first and the last residuals are increasing, i.e., $x_{i_0,0} \sqsubseteq_\mathrm{p} x_{i_1,0} \sqsubseteq_\mathrm{p} x_{i_2,0} \sqsubseteq_\mathrm{p} \cdots$ and $x_{i_0,k_0} \sqsubseteq_\mathrm{p} x_{i_1,k_1} \sqsubseteq_\mathrm{p} x_{i_2,k_2} \sqsubseteq_\mathrm{p} \cdots$. Now recall that, by Higman's Lemma, the sequence extension $((\Sigma_{d-1}^*)^*, \preceq_*)$ is a wqo since, by ind. hyp., $(\Sigma_{d-1}^*, \sqsubseteq_\mathrm{p})$ is a wqo. We may thus further extract an infinite subsequence that is increasing for $\preceq_*$ on the residuals, i.e., with $\langle x_{i_0,0}, x_{i_0,1}, \ldots, x_{i_0,k_0} \rangle \preceq_* \langle x_{i_1,0}, x_{i_1,1}, \ldots, x_{i_1,k_1} \rangle \preceq_* \langle x_{i_2,0}, x_{i_2,1}, \ldots, x_{i_2,k_2} \rangle \preceq_* \cdots$ With Lemma 6 we deduce $x_{i_0} \sqsubseteq_\mathrm{p} x_{i_1} \sqsubseteq_\mathrm{p} x_{i_2} \sqsubseteq_\mathrm{p} \cdots$. Hence $(\Sigma_d^*, \sqsubseteq_\mathrm{p})$ is a wqo. □

Remark 8. Thm. 7 and Prop. 5 prove that $\leq_\#$ is a wqo on configurations of PCSs, as we assumed in Sec. 2.3. There we also assumed that $\leq_\#$ is decidable. We can now see that it is in NLogSpace, since, in view of Prop. 5, one can check whether $x \leq_\# y$ by reading x and y simultaneously while guessing nondeterministically a factorization $z_1 a_1 \cdots z_\ell a_\ell$ of y, and checking that $z_i \in \Sigma_{a_i}^*$.

Fig. 2. Two trees in T_2

4 Applications of the Priority Embedding to Trees

In this section we show how tree orderings can be reflected into sequences over a priority alphabet. This serves two purposes. First, it illustrates the "power" of priority embeddings, giving a simple proof that strong tree embeddings form a wqo as a byproduct. Second, the reflection defined will subsequently be used in Sec. 6 to provide an encoding of ordinals that PCSs can manipulate "robustly."

4.1 Encoding Bounded Depth Trees

Given an alphabet Γ, the set of finite, ordered, unranked labeled trees (aka variadic terms) over Γ, noted $T(\Gamma)$, is the smallest set such that, if f is in Γ and $t_1, \ldots, t_n$ are $n \geq 0$ trees in $T(\Gamma)$, then the tree $f(t_1 \cdots t_n)$ is in $T(\Gamma)$. A *context* C is defined as usual as a tree with a single occurrence of a leaf labeled by a distinguished variable x. Given a context C and a tree t, we can form a tree $C[t]$ by plugging t instead of that x-labeled leaf.

Let d be a depth in $\mathbb{N}$ and $\bullet$ be a node label. We consider the set $T_d = T_d(\{\bullet\})$ of trees of depth at most d with $\bullet$ as single possible label; for instance, $T_0 = \{\bullet()\}$ contains a single tree, and the two trees shown in Fig. 2 are in T_2:

It is a folklore result that one can encode bounded depth trees into finite sequences using canonical factorizations. Here we present a natural variant that is rather well-suited for our constructions in Sec. 6. We encode trees of bounded depth using the mapping $s_d : T_{d+1} \to \Sigma_d^*$ defined by induction on d as

$$s_d(\bullet(t_1 \cdots t_n)) \stackrel{\text{def}}{=} \begin{cases} \varepsilon & \text{if } n = 0, \\ s_{d-1}(t_1)d \cdots s_{d-1}(t_n)d & \text{otherwise.} \end{cases} \tag{4}$$

For instance, if we fix $d = 1$, the left tree in Fig. 2 is encoded as "111" and the right one as "0011". Note that the encoding depends on the choice of d: for $d = 2$ we would have encoded the trees in Fig. 2 as "222" and "1122", respectively.

Not every string in Σ_d^* is the encoding of a tree according to s_d: for $-1 \leq a \leq d$, we let $P_a \stackrel{\text{def}}{=} (P_{a-1}\{a\})^*$ be the set of *proper encodings of height* a, with further $P_{-1} \stackrel{\text{def}}{=} \{\varepsilon\}$. Then $P \stackrel{\text{def}}{=} \bigcup_{a \leq d} P_a$ is the set of *proper* words in Σ_d^*. A proper word x is either empty or belongs to a unique P_a with $a = h(x)$, and has then a canonical factorization of the form $x = x_1 a \cdots x_m a$ with every x_j in P_{a-1}. Put differently, a non-empty $x = a_1 \cdots a_\ell$ is in P_a if and only if $a_\ell = h(x)$ and $a_{i+1} - a_i \leq 1$ for all $i < \ell$ (we say that x *has no jumps*: along proper words, priorities only increase smoothly, but can decrease sharply). For example, 02 is not proper (it has a jump) while 012 is proper; 233123401234 is proper too.

Given a depth a, we see that s_a is a bijection between T_{a+1} and P_a, with the inverse defined by

$$\tau(\varepsilon) \stackrel{\text{def}}{=} \bullet() \,, \qquad \tau(x = x_1 h(x) \cdots x_m h(x)) \stackrel{\text{def}}{=} \bullet(\tau(x_1) \cdots \tau(x_m)) \,. \tag{5}$$

4.2 Strong Tree Embeddings

One can provide a formal meaning to the notion of a wqo $(B, \preccurlyeq_B)$ being more powerful than another one $(A, \preccurlyeq_A)$ through *order reflections*, i.e. through the existence of a mapping $r\colon A \to B$ such that $r(x) \preccurlyeq_B r(y)$ implies $x \preccurlyeq_A y$ for all x, y in A. Observe that if B reflects A and $(B, \preccurlyeq_B)$ is a wqo, then $(A, \preccurlyeq_A)$ is necessarily a wqo. We show here that $(\Sigma_d^*, \sqsubseteq_{\mathrm{p}})$ reflects bounded-depth trees endowed with the strong tree-embedding relation.

Let t and t' be two trees in T_d. We say that t *strongly embeds* into t', written $t \sqsubseteq_T t'$, if it can be obtained from t' by deleting whole subtrees, i.e. $\sqsubseteq_T$ is the reflexive transitive closure of the relation $t \sqsubset_T^1 t' \stackrel{\text{def}}{\Leftrightarrow} t = C[\bullet(t_1 \cdots t_{i-1} t_{i+1} \cdots t_n)]$ and $t' = C[\bullet(t_1 \cdots t_{i-1} t_i t_{i+1} \cdots t_n)]$ for some context C and subtrees $t_1, \ldots, t_n$. Strong tree embeddings refine the *homeomorphic tree embeddings* used in Kruskal's Tree Theorem; in general they do not give rise to a wqo, but in the case of bounded depth trees they do. The two trees in Fig. 2 are *not* related by any homeomorphic tree embedding, and thus neither by strong tree embedding. See the full version for the proofs of the following results:

Proposition 9. *The map s_d is an order reflection from $(T_{d+1}, \sqsubseteq_T)$ to $(\Sigma_d^*, \sqsubseteq_p)$.*

Corollary 10. *For each d, $(T_d, \sqsubseteq_T)$ is a wqo.*

4.3 Further Applications

As stated in the introduction to this section, our main interest in strong tree embeddings is in connection with structural orderings of ordinals; see Sec. 6. Bounded depth trees are also used in the verification of infinite-state systems as a means to obtain decidability results, in particular for tree pattern rewriting systems [15] in XML processing, and, using elimination trees [see 21], for bounded-depth graphs used e.g. in the verification of ad-hoc networks [12], the π-calculus [22], and programs [5]. These applications consider *labeled* trees, which are dealt with thanks to a generalization of $\sqsubseteq_{\mathrm{p}}$ to pairs (a, w) where a is a priority and w a symbol from some wqo $(\Gamma, \leq)$; see the full version.

This generalization of $\sqsubseteq_{\mathrm{p}}$ also allows to treat another wqo on trees, the *tree minor* ordering, using the techniques of Gupta [16] to encode them in prioritized alphabets. The tree minor ordering is coarser than the homeomorphic embedding (e.g. in Fig. 2, the left tree is a minor of the right tree), but the upside is that trees of unbounded depth can be encoded into strings.

The exact complexity of verification problems in the aforementioned models is currently unknown [15, 12, 22, 5]. Our encoding suggests them to be $\mathbf{F}_{\varepsilon_0}$-complete. We hope to see PCS Reachability employed as a "master" problem for $\mathbf{F}_{\varepsilon_0}$, like LCS Reachability for $\mathbf{F}_{\omega^\omega}$, which is used in reductions instead of more difficult proofs based on Turing machines and Hardy computations.

5 Fast-Growing Upper Bounds

The verification of infinite-state systems and WSTSs in particular turns out to require astronomic computational resources expressed as *subrecursive functions* [20, 13] of the input size. We show in this section how to bound the complexity of the algorithms presented in Sec. 2.3 and classify the Reachability and Inevitability problems using *fast-growing complexity classes* [25].

5.1 Subrecursive Hierarchies

Throughout this paper, we use *ordinal terms* inductively defined by the following grammar

$$(\Omega \ni)\ \ \alpha, \beta, \gamma \ ::=\ 0 \mid \omega^{\alpha} \mid \alpha + \beta$$

where addition is associative, with 0 as the neutral element (the empty sum). Equivalently, we can then see a term other than 0 as a tree over the alphabet $\{+\}$; for instance the two trees in Fig. 2 represent 3 and $\omega^2 + 1$ respectively, when putting the ordinal terms under the form $\alpha = \sum_{i=1}^{k} \omega^{\alpha_i}$. Such a term is 0 if $k = 0$, otherwise a *successor* if $\alpha_k = 0$ and a *limit* otherwise. We often write 1 as short-hand for ω^0, and ω for ω^1. The symbol λ is reserved for limits.

We can associate a set-theoretic ordinal $o(\alpha)$ to each term α by interpreting $+$ as the direct sum operator and ω as $\mathbb{N}$; this gives rise to a well-founded quasi-ordering $\alpha < \beta \overset{\text{def}}{\Leftrightarrow} o(\alpha) < o(\beta)$. A term $\alpha = \sum_{i=1}^{k} \omega^{\alpha_i}$ is in *Cantor normal form* (CNF) if $\alpha_1 \geq \alpha_2 \geq \cdots \geq \alpha_k$ and each α_i is itself in CNF for $i = 1, \ldots, k$. Terms in CNF and set-theoretic ordinals below ε_0 are in bijection; it will however be convenient later in Sec. 6 to manipulate terms that are *not* in CNF.

With any limit term λ, we associate a *fundamental sequence* of terms $(\lambda_n)_{n\in\mathbb{N}}$

$$\begin{aligned} (\gamma + \omega^{\beta+1})_n &\overset{\text{def}}{=} \gamma + \omega^{\beta} \cdot n = \gamma + \overbrace{\omega^{\beta} + \cdots + \omega^{\beta}}^{n}\,, \\ (\gamma + \omega^{\lambda'})_n &\overset{\text{def}}{=} \gamma + \omega^{\lambda'_n}\,. \end{aligned} \tag{6}$$

This yields $\lambda_0 < \lambda_1 < \cdots < \lambda$ for any λ, with furthermore $\lambda = \lim_{n\in\mathbb{N}} \lambda_n$. For instance, $\omega_n = n$, $(\omega^{\omega})_n = \omega^n$, etc. Note that λ_n is in CNF when λ is.

We need to add a term ε_0 to Ω to represent the set-theoretic ε_0, i.e. the smallest solution of $x = \omega^x$. We take this term to be a limit term as well; we define the fundamental sequence for ε_0 by $(\varepsilon_0)_n \overset{\text{def}}{=} \Omega_n$, where for $n \in \mathbb{N}$, we use Ω_n as short-hand notation for the ordinal $\omega^{\omega^{\cdots^{\omega}}}\}n$ stacked ω's, i.e., for $\Omega_0 \overset{\text{def}}{=} 1$ and $\Omega_{n+1} \overset{\text{def}}{=} \omega^{\Omega_n}$.

Inner Recursion Hierarchies. Our main subrecursive hierarchy is the *Hardy hierarchy*. Given a monotone expansive unary function $h\colon \mathbb{N} \to \mathbb{N}$, it is defined as an ordinal-indexed hierarchy of unary functions $(h^{\alpha}\colon \mathbb{N} \to \mathbb{N})_{\alpha}$ through

$$h^0(n) \overset{\text{def}}{=} n\,, \qquad h^{\alpha+1}(n) \overset{\text{def}}{=} h^{\alpha}\big(h(n)\big)\,, \qquad h^{\lambda}(n) \overset{\text{def}}{=} h^{\lambda_n}(n)\,.$$

Observe that h^1 is simply h, and more generally h^{α} is the αth iterate of h, using diagonalisation to treat limit ordinals.

A case of particular interest is to choose the successor function $H(n) \stackrel{\text{def}}{=} n+1$ for h. Then the *fast growing hierarchy* $(F_\alpha)_\alpha$ can be defined by $F_\alpha \stackrel{\text{def}}{=} H^{\omega^\alpha}$, resulting in $F_0(n) = H^1(n) = n+1$, $F_1(n) = H^\omega(n) = H^n(n) = 2n$, $F_2(n) = H^{\omega^2}(n) = 2^n n$ being exponential, $F_3 = H^{\omega^3}$ being non-elementary, $F_\omega = H^{\omega^\omega}$ being an Ackermannian function, F_{ω^k} a k-Ackermannian function, and $F_{\varepsilon_0} = H^{\varepsilon_0} \circ H$ a function whose totality is not provable in Peano arithmetic [13].

Fast-Growing Complexity Classes. Our intention is to establish the "F_{ε_0} completeness" of verification problems on PCSs. In order to make this statement more precise, we define the class $\mathbf{F}_{\varepsilon_0}$ as a specific instance of the *fast-growing complexity classes* defined for $\alpha \geq 3$ by [see 25, App. B]

$$\mathbf{F}_\alpha \stackrel{\text{def}}{=} \bigcup_{p \in \bigcup_{\beta<\alpha} \mathscr{F}_\beta} \text{DTime}(F_\alpha(p(n))) \,, \qquad \mathscr{F}_\alpha = \bigcup_{c<\omega} \text{FDTime}(F_\alpha^c(n)) \,, \tag{7}$$

where the class of functions $\mathscr{F}_\alpha$ as defined above is the αth level of the *extended Grzegorczyk hierarchy* [20] when $\alpha \geq 2$; in particular, $\bigcup_{\alpha<\varepsilon_0} \mathscr{F}_\alpha$ is exactly the set of ordinal-recursive (aka "provably recursive") functions [13].

The complexity classes $\mathbf{F}_\alpha$ are naturally equipped with $\bigcup_{\beta<\alpha} \mathscr{F}_\beta$ as classes of reductions. For instance, $\mathscr{F}_2$ is the set of elementary functions, and $\mathbf{F}_3$ the class of problems with a tower of exponents of height bounded by some elementary function of the input as an upper bound.[3]

5.2 Complexity Upper Bounds

Recall that an alternative characterization of a wqo $(X, \preccurlyeq)$ is that any sequence $x_0, x_1, x_2, \ldots$ over X verifying $x_i \not\preccurlyeq x_j$ for all $i < j$ is necessarily finite. Such sequences are called *bad*, and in order to bound the complexity of the algorithms from Thm. 3, we can bound the lengths of bad sequences over the wqo $(Conf_{\mathcal{S}}, \leq_\#)$ using the *Length Function Theorem* of [24]; see the full version for details:

Theorem 11 (Complexity of PCS Verification). *Reachability and Inevitability of PCSs are in* $\mathbf{F}_{\varepsilon_0}$.

6 Hardy Computations by PCSs

In this section we show how PCSs can weakly compute the Hardy functions H^α and their inverses for all ordinals α below Ω, which is the key ingredient for Thm. 15. For this, we develop (Sec. 6.1) encodings $s(\alpha) \in \Sigma_d^*$ for ordinals $\alpha \in \Omega_d$ and show how PCSs can compute with these codes, e.g. build the code for λ_n from the code of a limit λ. This is used (Sec. 6.2) to design PCSs that "*weakly compute*" H^α and $(H^\alpha)^{-1}$ in the sense of Def. 13 below.

[3] Note that, at such high complexities, the usual distinctions between deterministic vs. nondeterministic, or time-bounded vs. space-bounded computations become irrelevant.

6.1 Encoding Ordinals

Our encoding of ordinal terms as strings in Σ_d^* is exactly the encoding of trees presented in Sec. 4. For $0 \leq a \leq d$, we use the following equation to define the language $P_a \subseteq \Sigma_d^*$ of *proper encodings*, or just *codes*:

$$P_a \stackrel{\text{def}}{=} \varepsilon + P_a P_{a-1} a\,, \qquad\qquad P_{-1} \stackrel{\text{def}}{=} \varepsilon\,. \tag{8}$$

Let $P = P_{-1} + P_0 + \cdots + P_d$. Each P_a (and then P itself) is a regular language, with $P_a = (P_{a-1}a)^*$ as in Sec. 4; for instance, $P_0 = 0^*$.

Decompositions. A code x is either the empty word ε, or belongs to a unique P_a. If $x \in P_a$ is not empty, it has a unique factorization $x = yza$ according to (8) with $y \in P_a$ and $z \in P_{a-1}$. The factor $z \in P_{a-1}$ in $x = yza$ can be developed further, as long as $z \neq \varepsilon$: a non-empty code $x \in P_d$ has a unique factorization as $x = y_d\, y_{d-1} \ldots y_a\, a^\frown d$ with $y_i \in P_i$ for $i = a, \ldots, d$, and where for $0 \leq a \leq b$, we write $a^\frown b$ for the *staircase* word $a(a+1)\cdots(b-1)b$, letting $a^\frown b = \varepsilon$ when $a > b$. We call this the *decomposition* of x. Note that the value of a is obtained by looking for the maximal suffix of x that is a staircase word. For example, $x = 23312340121234 \in P_4$ is a code and decomposes as

$$x = \overbrace{2331234}^{y_4}\overbrace{\varepsilon}^{y_3}\overbrace{012}^{y_2}\overbrace{\varepsilon}^{y_1}\overbrace{1234}^{1^\frown 4}.$$

Ordinal Encoding. Following the tree encoding of Sec. 4, with a code $x \in P$, we associate an ordinal term $\eta(x)$ given by

$$\eta(\varepsilon) \stackrel{\text{def}}{=} 0\,, \qquad\qquad \eta(yza) \stackrel{\text{def}}{=} \eta(y) + \omega^{\eta(z)}\,, \tag{9}$$

where $x = yza$ is the factorization according to (8) of $x \in P_a \setminus \{\varepsilon\}$. For example, $\eta(a) = \omega^0 = 1$ for all $a \in \Sigma_d$, $\eta(012) = \eta(234) = \omega^\omega$, and more generally $\eta(a^\frown b) = \Omega_{b-a}$. One sees that $\eta(x) < \Omega_{a+1}$ when $x \in P_a$.

The *decoding* function $\eta\colon P \to \Omega_{d+1}$ is onto (or surjective) but it is not bijective. However, it is a bijection between P_a and Ω_{a+1} for any $a \leq d$. Its converse is the level-a *encoding* function $s_a\colon \Omega_{a+1} \to P_a$, defined with

$$s_a\Big(\sum_{i=1}^{p} \gamma_i\Big) \stackrel{\text{def}}{=} s_a(\gamma_1)\cdots s_a(\gamma_p)\,, \qquad\qquad s_a(\omega^\alpha) \stackrel{\text{def}}{=} s_{a-1}(\alpha)\, a\,.$$

Thus $s_a(0) = s_a(\sum \emptyset) = \varepsilon$ and, for example,

$$\begin{aligned} s_5(1) &= 5\,, & s_5(3) &= 555\,, & s_5(\omega) &= 45\,,\\ s_5(\omega^3) &= 4445\,, & s_5(\omega^\omega) &= 345\,, & s_5(\omega^{\omega^\omega}) &= 2345\,,\\ s_5(\omega^3 + \omega^2) &= 4445445\,, & s_5(\omega \cdot 3) &= 454545\,. \end{aligned}$$

We may omit the subscript when $a = d$, e.g. writing $s(1) = d$.

o :	334545$	the *ordinal* term $\omega^{\omega^2}+\omega^\omega$
c :	0000$	the *counter* value 4
t :	$	the *temporary* storage

Fig. 3. Channels for Hardy computations

Successors and Limits. Let $x = y_d\, y_{d-1} \dots y_a\, a^\frown d$ be the decomposition of $x \in P_d \setminus \varepsilon$. By (9), x encodes a successor ordinal $\eta(x) = \beta + 1$ iff $a = d$, i.e., if x ends with two d's (or has length 1). Since then $\beta = \eta(y_d \dots y_a)$, one obtains the "predecessor of x" by removing the final d.

If $a < d$, x encodes a limit λ. Combining (6) and (9), one obtains the encoding $(x)_n$ of λ_n with

$$(x)_n = y_d\, y_{d-1} \dots y_{a+1} \big(y_a(a+1)\big)^n (a+2)^\frown d\,. \tag{10}$$

E.g., with $d = 5$, decomposing $x = 333345 = s(\omega^{\omega^4})$ gives $a = 3$, $x = y_5 y_4 y_3 3^\frown 5$, with $y_3 = 333$ and $y_5 = y_4 = \varepsilon$. Then $(x)_n = (3334)^n 5$, agreeing with, e.g. $s(\omega^{\omega^3 \cdot 2}) = 333433345$.

Robustness. Translated to ordinals, Prop. 9 means that, whenever $x \leq_\# x'$ for $x, x' \in P_a$, then the corresponding ordinal $\eta(x)$ will be "structurally" smaller than $\eta(x')$. This in turn yields that the corresponding Hardy function $H^{\eta(x)}$ grows at most as fast as $H^{\eta(x')}$; see the full version for details:

Proposition 12 (Robustness). *Let $a \geq 0$ and $x, x' \in P_a$. If $x \leq_\# x'$ then $H^{\eta(x)}(n) \leq H^{\eta(x')}(n')$ for all $n \leq n'$ in $\mathbb{N}$.*

6.2 Robust Hardy Computations in PCSs

Our PCSs for robust Hardy computations use three channels (see Fig. 3), storing (codes for) a pair α, n on channels o (for "ordinal") and c (for "counter"), and employ an extra channel, t, for "temporary" storage. Instead of Σ_d, we use Σ_{d+1} with $d{+}1$ used as a position marker and written \$ for clarity: each channel always contains a single occurrence of \$.

Definition 13. *A* weak Hardy computer *for Ω_{d+1} is a $(d+1)$-PCS S with channels $\mathtt{Ch} = \{\mathtt{o}, \mathtt{c}, \mathtt{t}\}$ and two distinguished states p_{beg} and p_{end} such that:*

$$\begin{aligned} &\textit{if } (p_{beg}, x\$, y\$, z\$) \xrightarrow{*}_\# (p_{end}, u, v, w) \\ &\quad \textit{then } x \in P_d, y \in 0^+, z = \varepsilon \textit{ and } u, v, w \in \Sigma_d^*\$\,, \end{aligned} \tag{safety}$$

$$\begin{aligned} &\textit{if } (p_{beg}, s(\alpha)\$, 0^n\$, \$) \xrightarrow{*}_\# (p_{end}, s(\beta)\$, 0^m\$, \$) \\ &\quad \textit{then } H^\alpha(n) \geq H^\beta(m)\,. \end{aligned} \tag{robustness}$$

Furthermore S is complete *if for any $\alpha < \Omega_{d+1}$ and $n > 0$, $(p_{beg}, s(\alpha)\$, 0^n\$, \$) \xrightarrow{*}_\# (p_{end}, \$, 0^m\$, \$)$ for $m = H^\alpha(n)$, and it is* inv-complete *if $(p_{beg}, \$, 0^m\$, \$) \xrightarrow{*}_\# (p_{end}, s(\alpha)\$, 0^n\$, \$)$.*

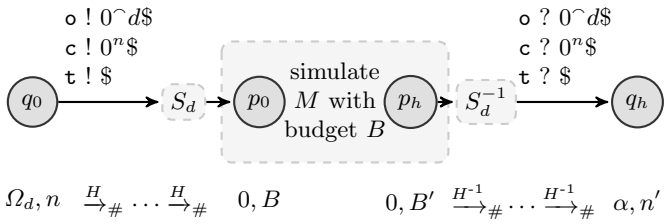


Fig. 4. Schematics for Thm. 15

In the full version we prove the following:

Lemma 14 (PCSs weakly compute Hardy functions). *For every $d \in \mathbb{N}$, there exists a weak Hardy computer S_d for Ω_{d+1} that is complete, and a weak S_d^{-1} that is inv-complete. Furthermore S_d and S_d^{-1} can be generated in* LogSpace *from d.*

6.3 Wrapping It Up

With the above weak Hardy computers, we have the essential gadgets required for our reductions. The wrapping-up is exactly as in [17, 27] (with a different encoding and a different machine model) and will only be sketched.

Theorem 15 (Verifying PCSs is Hard). *Reachability and Termination of PCSs are $\mathbf{F}_{\varepsilon_0}$-hard.*

Proof. We exhibit a LogSpace reduction from the halting problem of a Turing machine M working in F_{ε_0} space to the Reachability problem in a PCS. We assume wlog. M to start in a state p_0 with an empty tape and to have a single halting state p_h that can only be reached after clearing the tape.

Figure 4 depicts the PCS S we construct for the reduction. Let $n \stackrel{\text{def}}{=} |M|$ and $d \stackrel{\text{def}}{=} n+1$. A run in S from the initial configuration to the final one goes through three stages:

1. The first stage robustly computes $F_{\varepsilon_0}(|M|) = H^{\Omega_d}(n)$ by first writing $s(\Omega_d)$\$, i.e. $0^\frown d$\$, on `o`, 0^n\$ on `c`, and \$ on `t`, then by using S_d to perform forward Hardy steps; thus upon reaching state p_0, `o` and `t` contain \$ and `c` encodes a budget $B \leq F_{\varepsilon_0}(|M|)$.
2. The central component simulates M over `c` where the symbols 0 act as blanks—this is easily done by cycling through the channel contents to simulate the moves of the head of M on its tape. Due to superseding steps, the outcome upon reaching p_h is that `c` contains $B' \leq B$ symbols 0.
3. The last stage robustly computes $(F_{\varepsilon_0})^{-1}(B')$ by running S_d^{-1} to perform backward Hardy steps. This leads to `o` containing the encoding of some ordinal α and `c` of some n', but we empty these channels and check that $\alpha = \Omega_d$ and $n' = n$ before entering state q_h.

Because $H^{\Omega_d}(n) \geq B \geq B' \geq H^{\alpha}(n') = H^{\Omega_d}(n)$, all the inequalities are actually equalities, and the simulation of M in stage 2 has necessarily employed reliable steps. Hence, M halts if and only if $(q_h, \varepsilon, \varepsilon, \varepsilon)$ is reachable from $(q_0, \varepsilon, \varepsilon, \varepsilon)$ in S.

The case of (non-)Termination is similar, but employs a *time* budget in a separate channel in addition to the space budget, in order to make sure that the simulation of M terminates in all cases, and leads to a state q_h that is the only one from which an infinite run can start in S. □

7 Concluding Remarks

We introduced Priority Channel Systems, a natural model for protocols and programs with differentiated, prioritized asynchronous communications, and showed how they give rise to well-structured systems with decidable model-checking problems.

We showed that Reachability and Termination for PCSs are $\mathbf{F}_{\varepsilon_0}$-complete, and we expect our techniques to be transferable to other models, e.g. models based on wqos on bounded-depth trees or graphs, whose complexity has not been analyzed [15, 12, 22, 5]. This is part of our current research agenda on complexity for well-structured systems [25].

In spite of their enormous worst-case complexity, we expect PCSs to be amenable to regular model checking techniques *à la* [4, 6]. This requires investigating the algorithmics of upward- and downward-closed sets of configurations wrt. the priority ordering. These sets, which are always regular, seem promising since $\sqsubseteq_{\mathrm{p}}$ shares some good properties with the better-known subword ordering, e.g. the upward- or downward-closure of a sequence $x \in \Sigma_d^*$ can be represented by a DFA with $|x|$ states.

References

1. Abdulla, P.A., Atig, M.F., Cederberg, J.: Timed lossy channel systems. In: FST&TCS 2012. LIPIcs, vol. 18, pp. 374–386. Leibniz-Zentrum für Informatik (2012)
2. Abdulla, P.A., Čerāns, K., Jonsson, B., Tsay, Y.K.: Algorithmic analysis of programs with well quasi-ordered domains. Inform. and Comput. 160(1-2), 109–127 (2000)
3. Abdulla, P.A., Deneux, J., Ouaknine, J., Worrell, J.: Decidability and complexity results for timed automata via channel machines. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1089–1101. Springer, Heidelberg (2005)
4. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. Inform. and Comput. 127(2), 91–101 (1996)
5. Bansal, K., Koskinen, E., Wies, T., Zufferey, D.: Structural counter abstraction. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 62–77. Springer, Heidelberg (2013)
6. Boigelot, B., Godefroid, P.: Symbolic verification of communication protocols with infinite state spaces using QDDs. Form. Methods in Syst. Des. 14(3), 237–255 (1999)
7. Bouajjani, A., Habermehl, P.: Symbolic reachability analysis of FIFO-channel systems with nonregular sets of configurations. Theor. Comput. Sci. 221(1-2), 211–250 (1999)

8. Bouyer, P., Markey, N., Ouaknine, J., Schnoebelen, P., Worrell, J.: On termination and invariance for faulty channel machines. Form. Asp. Comput. 24(4-6), 595–607 (2012)
9. Cécé, G., Finkel, A.: Verification of programs with half-duplex communication. Inform. and Comput. 202(2), 166–190 (2005)
10. Cécé, G., Finkel, A., Purushothaman Iyer, S.: Unreliable channels are easier to verify than perfect channels. Inform. and Comput. 124(1), 20–31 (1996)
11. Chambart, P., Schnoebelen, P.: The ordinal recursive complexity of lossy channel systems. In: LICS 2008, pp. 205–216. IEEE Press (2008)
12. Delzanno, G., Sangnier, A., Zavattaro, G.: Parameterized verification of ad hoc networks. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 313–327. Springer, Heidelberg (2010)
13. Fairtlough, M., Wainer, S.S.: Hierarchies of provably recursive functions. In: Handbook of Proof Theory, ch. III, pp. 149–207. Elsevier (1998)
14. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. 256(1-2), 63–92 (2001)
15. Genest, B., Muscholl, A., Serre, O., Zeitoun, M.: Tree pattern rewriting systems. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 332–346. Springer, Heidelberg (2008)
16. Gupta, A.: A constructive proof that trees are well-quasi-ordered under minors. In: Nerode, A., Taitslin, M.A. (eds.) LFCS 1992. LNCS, vol. 620, pp. 174–185. Springer, Heidelberg (1992)
17. Haddad, S., Schmitz, S., Schnoebelen, P.: The ordinal-recursive complexity of timed-arc Petri nets, data nets, and other enriched nets. In: LICS 2012, pp. 355–364. IEEE Press (2012)
18. Kurucz, A.: Combining modal logics. In: Handbook of Modal Logics, ch. 15, pp. 869–926. Elsevier (2006)
19. Lasota, S., Walukiewicz, I.: Alternating timed automata. ACM Trans. Comput. Logic 9(2) (2008)
20. Löb, M., Wainer, S.: Hierarchies of number theoretic functions. I. Arch. Math. Logic 13, 39–51 (1970)
21. Ossona de Mendez, P., Nešetřil, J.: Sparsity, ch. 6. Bounded height trees and tree-depth, pp. 115–144. Springer (2012)
22. Meyer, R.: On boundedness in depth in the π-calculus. In: Ausiello, G., Karhumäki, J., Mauri, G., Ong, L. (eds.) IFIP TCS 2008. IFIP, vol. 273, pp. 477–489. Springer, Boston (2008)
23. Ouaknine, J., Worrell, J.: On the decidability and complexity of Metric Temporal Logic over finite words. Logic. Meth. Logic. Meth. in Comput. Sci. 3(1), 1–27 (2007)
24. Schmitz, S., Schnoebelen, P.: Multiply-recursive upper bounds with Higman's lemma. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 441–452. Springer, Heidelberg (2011)
25. Schmitz, S., Schnoebelen, P.: Algorithmic aspects of WQO theory. Lecture notes (2012), `http://cel.archives-ouvertes.fr/cel-00727025`
26. Schnoebelen, P.: Lossy counter machines decidability cheat sheet. In: Kučera, A., Potapov, I. (eds.) RP 2010. LNCS, vol. 6227, pp. 51–75. Springer, Heidelberg (2010)
27. Schnoebelen, P.: Revisiting Ackermann-hardness for lossy counter machines and reset Petri nets. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 616–628. Springer, Heidelberg (2010)
28. Schütte, K., Simpson, S.G.: Ein in der reinen Zahlentheorie unbeweisbarer Satz über endliche Folgen von natürlichen Zahlen. Arch. Math. Logic 25(1), 75–89 (1985)

Reachability Probabilities of Quantum Markov Chains

Shenggang Ying, Yuan Feng, Nengkun Yu, and Mingsheng Ying

Tsinghua University, China
University of Technology, Sydney, Australia
{Shenggang.Ying,Yuan.Feng,Mingsheng.Ying}@uts.edu.au

Abstract. This paper studies three kinds of long-term behaviour, namely reachability, repeated reachability and persistence, of quantum Markov chains (qMCs). As a stepping-stone, we introduce the notion of bottom strongly connected component (BSCC) of a qMC and develop an algorithm for finding BSCC decompositions of the state space of a qMC. As the major contribution, several (classical) algorithms for computing the reachability, repeated reachability and persistence probabilities of a qMC are presented, and their complexities are analysed.

Keywords: quantum Markov chains, reachability, persistence.

1 Introduction

Verification problems of quantum systems are emerging from quantum physics, quantum communication and quantum computation. For example, verification has been identified by physicists as one of the major short-term goals of quantum simulation [4]. Some effective verification techniques for quantum cryptographic protocols have recently been developed [12], [6], based on either quantum process algebras [15], [11], [8], [9] or quantum model-checking [13]. Also, several methods for verifying quantum programs [20] have been proposed, including quantum weakest preconditions [7] and quantum Floyd-Hoare logic [23].

A quantum Markov chain (qMC) is a quantum generalisation of Markov chain (MC) where, roughly speaking, the state space is a Hilbert space, and the transition probability matrix of a MC is replaced by a super-operator, which is a mathematical formalism of the discrete-time evolution of (open) quantum systems. qMCs have been widely employed as a mathematical model of quantum noise in physics [10] and as a model of communication channels in quantum information theory [17]. A special class of qMCs, namely quantum walks, has been successfully used in design and analysis of quantum algorithms [1]. Recently, the authors [24] introduced a model of concurrent quantum programs in terms of qMCs as a quantum extension of Hart-Sharir-Pnueli's Markov chain model of probabilistic concurrent programs [14]. This paper considers the verification problem of qMCs.

Reachability analysis is at the center of verification and model-checking of both classical and probabilistic systems. Reachability of quantum systems was first studied by physicists [19] within the theme of quantum control , but they only considered states reachable in a single step of evolution. In [24], reachability of qMCs was considered, and it was used in termination checking of concurrent quantum programs. However,

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 334–348, 2013.

reachability studied in [24] can be properly described as *qualitative* reachability because only algorithms for computing reachable subspaces but not reachability probabilities were developed. This paper is a continuation of [24] and aims at *quantitative* reachability analysis for qMCs. More precisely, the main purpose of this paper is to develop (classical) algorithms for computing the reachability, repeated reachability and persistence probabilities of qMCs.

Reachability analysis techniques for classical MCs heavily depends on algorithms for graph-reachability problems, in particular for finding bottom strongly connected components (BSCCs) of the underlying graph of a MC (see [2, Section 10.1.2]). Such algorithms have been intensively studied by the graph algorithms community since early 1970's (see [5, Part VI]; [22]), and are ready to be directly adopted in reachability analysis of MCs. However, we don't have the corresponding algorithms for qMCs in hands and have to start from scratch. So, in order to conduct reachability analysis for qMCs we introduce the notion of BSCC and develop an algorithm for finding BSCC decomposition for qMCs in this paper. Interestingly, there are some essential differences between BSCCs in the classical and quantum cases. For example, BSCC decomposition of a qMC is unnecessary to be unique. Also, classical algorithms for finding BSCCs like depth-first search cannot be directly generalised to qMCs. Instead, it requires very different ideas to develop algorithms for finding BSCCs of qMCs, appealing to matrix operation algorithms [5, Chapter 28] through matrix representation of super-operators. The major challenge in dealing with quantum BSCCs, which would not arise in classical BSCCs at all, is to maintain the linear algebraic structure underpinning quantum systems. We believe that these results for quantum BSCCs obtained in this paper are also of independent significance.

This paper is organised as follows. The preliminaries are presented in Sec. 2; in particular we recall the notion of qMC and define the graph structure of a qMC. The notion of BSCC of a qMC is introduced in Sec. 3, where a characterisation of quantum BSCC is given in terms of the fixed points of super-operators, and an algorithm for checking whether a subspace of the state Hilbert space of a qMC is a BSCC is given. In Sec. 4, we define the notion of transient subspace of a qMC and show that the state space of a qMC can be decomposed into the direct sum of a transient subspace and a family of BSCCs. Furthermore, it is proved that although such a decomposition is not unique, the dimensions of its components are fixed. In particular, an algorithm for constructing BSCC decomposition of qMCs is found. With the preparation in Secs. 3 and 4, we examine reachability of a qMC in Sec. 5, where an algorithm for computing reachability probability is presented. An algorithm for computing repeated reachability and persistence probabilities is finally developed in Sec. 6. Sec. 7 is a brief conclusion.

2 Quantum Markov Chains and Their Graph Structures

2.1 Basics of Quantum Theory

For convenience of the reader, we recall some basic notions from quantum theory; for details we refer to [17]. The state space of a quantum system is a Hilbert space. In this paper, we only consider a finite-dimensional Hilbert space $\mathcal{H}$, which is just a finite-dimensional complex vector space with inner product. The inner product of two vectors

$|\phi\rangle, |\psi\rangle \in \mathcal{H}$ is denoted by $\langle\phi|\psi\rangle$. A pure quantum state is a normalised vector $|\phi\rangle$ in $\mathcal{H}$ with $\langle\phi|\phi\rangle = 1$. We say that two vectors $|\phi\rangle$ and $|\psi\rangle$ are orthogonal, written $|\phi\rangle \perp |\psi\rangle$, if $\langle\phi|\psi\rangle = 0$. A mixed state is represented by a density operator, i.e. a positive operator ρ on $\mathcal{H}$ with $tr(\rho) = 1$, or equivalently a positive semi-definite and trace-one $n \times n$ matrix if $\dim \mathcal{H} = n$. In particular, for each pure state $|\psi\rangle$, there is a corresponding density operator $\psi = |\psi\rangle\langle\psi|$. For simplicity, we often use pure state $|\psi\rangle$ and density operator ψ interchangeably. A positive operator ρ is called a partial density operator if trace $\mathrm{tr}(\rho) \leq 1$. The set of partial density operators on $\mathcal{H}$ is denoted by $\mathcal{D}(\mathcal{H})$. The support $\mathrm{supp}(\rho)$ of a partial density operator $\rho \in \mathcal{D}(\mathcal{H})$ is defined to be the space spanned by the eigenvectors of ρ with non-zero eigenvalues. The set of all (bounded) operators on $\mathcal{H}$, i.e. $d \times d$ complex matrices with $d = \dim \mathcal{H}$, is denoted by $\mathcal{B}(\mathcal{H})$.

For any set V of vectors in $\mathcal{H}$, we write $\mathrm{span}V$ for the subspace of $\mathcal{H}$ spanned by V; that is, it consists of all finite linear combinations of vectors in V. Two subspaces X and Y of $\mathcal{H}$ are said to be orthogonal, written $X \perp Y$, if $|\phi\rangle \perp |\psi\rangle$ for any $|\phi\rangle \in X$ and $|\psi\rangle \in Y$. The ortho-complement $X^\perp$ of a subspace X of $\mathcal{H}$ is the subspace of vectors orthogonal to all vectors in X. An operator P is called the projection onto a subspace X if $P|\psi\rangle = |\psi\rangle$ for all $|\psi\rangle \in X$ and $P|\psi\rangle = 0$ for all $|\psi\rangle \in X^\perp$. We write P_X for the projection onto X. According to the theory of quantum measurements, for any density operator ρ, trace $tr(P_X\rho)$ is the probability that the mixed state ρ lies in subspace X. Let $\{X_k\}$ be a family of subspaces of $\mathcal{H}$. Then the join of $\{X_k\}$ is defined by

$$\bigvee_k X_k = \mathrm{span}(\bigcup_k X_k).$$

In particular, we write $X \vee Y$ for the join of two subspaces X and Y. It is easy to see that $\bigvee_k X_k$ is the smallest subspace of $\mathcal{H}$ that contains all X_k.

Composed quantum systems are modeled by tensor products. If a quantum system consists of two subsystems with state spaces $\mathcal{H}_1$ and $\mathcal{H}_2$, then its state space is $\mathcal{H} = \mathcal{H}_1 \otimes \mathcal{H}_2$, which is the Hilbert space spanned by vectors $|\psi_1\rangle|\psi_2\rangle = |\psi_1\rangle \otimes |\psi_2\rangle$ with $|\psi_1\rangle \in \mathcal{H}_1$ and $|\psi_2\rangle \in \mathcal{H}_2$. For any operators A_1 on $\mathcal{H}_1$ and A_2 on $\mathcal{H}_2$, their tensor product $A_1 \otimes A_2$ is defined by $(A_1 \otimes A_2)(|\psi_1\rangle|\psi_2\rangle) = (A_1|\psi_1\rangle) \otimes (A_2|\psi_2\rangle)$ for all $|\psi_1\rangle \in \mathcal{H}_1$ and $|\psi_2\rangle \in \mathcal{H}_2$ together with linearity.

The evolution of a closed quantum system is described as a unitary operator, i.e. an operator U on $\mathcal{H}$ with $U^\dagger U = UU^\dagger = I$, where I is the identity on $\mathcal{H}$. A pure state $|\phi\rangle$ becomes $U|\phi\rangle$ after this unitary evolution U, while a mixed state ρ becomes $U\rho U^\dagger$. The dynamics of an open quantum system is described by a super-operator, i.e. a linear map $\mathcal{E}$ from the space of linear operators on $\mathcal{H}$ into itself, satisfying the following conditions:

1. $\mathrm{tr}[\mathcal{E}(\rho)] \leq \mathrm{tr}(\rho)$ for all $\rho \in \mathcal{D}(\mathcal{H})$, with equality for trace-preserving $\mathcal{E}$;
2. Complete positivity: for any extra Hilbert space $\mathcal{H}_R$, $(\mathcal{I}_R \otimes \mathcal{E})(A)$ is positive provided A is a positive operator on $\mathcal{H}_R \otimes \mathcal{H}$, where $\mathcal{I}_R$ is the identity map on the space of linear operators on $\mathcal{H}_R$.

In this paper, we only consider trace-preserving super-operators. Each super-operator has a Kraus operator-sum representation: $\mathcal{E} = \sum_i E_i \cdot E_i^\dagger$, or more precisely

$$\mathcal{E}(\rho) = \sum E_i \rho E_i^\dagger$$

for all $\rho \in \mathcal{D}(\mathcal{H})$, where E_i are operators on $\mathcal{H}$ such that $\sum_i E_i^\dagger E_i = I$.

2.2 Quantum Markov Chains

Now we are ready to introduce the notion of quantum Markov chain. Recall that a Markov chain is a pair $\langle S, P\rangle$, where S is a finite set of states, and P is a matrix of transition probabilities, i.e. a mapping $P : S \times S \to [0,1]$ such that $\sum_{t\in S} P(s,t) = 1$ for every $s \in S$, where $P(s,t)$ is the probability of going from s to t. A quantum Markov chain is a quantum generalisation of a Markov chain where the state space of a Markov chain is replaced by a Hilbert space and its transition matrix is replaced by a super-operator.

Definition 1. *A quantum Markov chain is a pair $\mathcal{G} = \langle \mathcal{H}, \mathcal{E}\rangle$, where $\mathcal{H}$ is a finite-dimensional Hilbert space, and $\mathcal{E}$ is a super-operator on $\mathcal{H}$.*

The behaviour of a quantum Markov chain can be described as follows: if currently the process is in a mixed state ρ, then it will be in state $\mathcal{E}(\rho)$ in the next step. Both ρ and $\mathcal{E}(\rho)$ can be written as statistical ensembles:

$$\rho = \sum_i p_i |\phi_i\rangle\langle\phi_i|, \qquad \mathcal{E}(\rho) = \sum_j q_j |\psi_j\rangle\langle\psi_j|,$$

where $p_i, q_j \geq 0$ for all i, j, and $\sum_i p_i = \sum_j q_j = 1$. So, super-operator $\mathcal{E}$ can be understood as an operation that transfers statistical ensemble $\{(p_i, |\phi_i\rangle)\}$ to $\{(q_j, |\psi_j\rangle)\}$. In this way, a quantum Markov chain can be seen as a generalisation of a Markov chain.

2.3 Graphs in Quantum Markov Chains

There is a natural graph structure underlying a quantum Markov chain. This can be seen clearly by introducing adjacency relation in it. To this end, we first introduce an auxiliary notion. The image of a subspace X of $\mathcal{H}$ under a super-operator $\mathcal{E}$ is defined to be

$$\mathcal{E}(X) = \bigvee_{|\psi\rangle\in X} \mathrm{supp}(\mathcal{E}(\psi)).$$

Intuitively, $\mathcal{E}(X)$ is the subspace of $\mathcal{H}$ spanned by the images under $\mathcal{E}$ of states in X.

Definition 2. *Let $\mathcal{G} = \langle \mathcal{H}, \mathcal{E}\rangle$ be a quantum Markov chain, and let $|\varphi\rangle$ and $|\psi\rangle$ be pure states and ρ and σ mixed states in $\mathcal{H}$. Then*

1. *$|\varphi\rangle$ is adjacent to $|\psi\rangle$ in $\mathcal{G}$, written $|\psi\rangle \to |\varphi\rangle$, if $|\varphi\rangle \in \mathcal{E}(X_\psi)$, where $X_\psi = \mathrm{span}\{|\psi\rangle\}$.*
2. *$|\varphi\rangle$ is adjacent to ρ, written $\rho \to |\varphi\rangle$, if $|\varphi\rangle \in \mathcal{E}(\mathrm{supp}(\rho))$.*
3. *σ is adjacent to ρ, written $\rho \to \sigma$, if $\mathrm{supp}(\sigma) \subseteq \mathcal{E}(\mathrm{supp}(\rho))$.*

Definition 3. 1. *A sequence $\pi = \rho_0 \to \rho_1 \to \cdots \to \rho_n$ of adjacent density operators in a quantum Markov chain $\mathcal{G}$ is called a path from ρ_0 to ρ_n in $\mathcal{G}$, and its length is $|\pi| = n$.*
2. *For any density operators ρ and σ, if there is a path from ρ to σ then we say that σ is reachable from ρ in $\mathcal{G}$.*

Definition 4. *Let $\mathcal{G} = \langle \mathcal{H}, \mathcal{E} \rangle$ be a quantum Markov chain. For any $\rho \in \mathcal{D}(\mathcal{H})$, its reachable space in $\mathcal{G}$ is*

$$\mathcal{R}_{\mathcal{G}}(\rho) = \text{span}\{|\psi\rangle \in \mathcal{H} : |\psi\rangle \text{ is reachable from } \rho \text{ in } \mathcal{G}\}.$$

The following lemma is very useful for our later discussion.

Lemma 1. *1. (Transitivity of reachability) For any $\rho, \sigma \in \mathcal{D}(\mathcal{H})$, if $\text{supp}(\rho) \subseteq \mathcal{R}_{\mathcal{G}}(\sigma)$, then $\mathcal{R}_{\mathcal{G}}(\rho) \subseteq \mathcal{R}_{\mathcal{G}}(\sigma)$.*

2. [24, Theorem 1] If $d = \dim \mathcal{H}$, then for any $\rho \in \mathcal{D}(\mathcal{H})$, we have

$$\mathcal{R}_{\mathcal{G}}(\rho) = \bigvee_{i=0}^{d-1} \text{supp}(\mathcal{E}^i(\rho)). \tag{1}$$

3 Bottom Strongly Connected Components

3.1 Basic Definitions

The notion of bottom strongly connected component plays an important role in model checking Markov chains. In this section, we extend this notion to the quantum case. We first introduce an auxiliary notation. Let X be a subspace of a Hilbert space, and let $\mathcal{E}$ be a super-operator on $\mathcal{H}$. Then the restriction of $\mathcal{E}$ on X is defined to be super-operator $\mathcal{E}|_X$ with

$$\mathcal{E}|_X(\rho) = P_X \mathcal{E}(\rho) P_X$$

for all $\rho \in \mathcal{D}(X)$, where P_X is the projection onto X.

Definition 5. *Let $\mathcal{G} = \langle \mathcal{H}, \mathcal{E} \rangle$ be a quantum Markov chain. A subspace X of $\mathcal{H}$ is called strongly connected in $\mathcal{G}$ if for any $|\varphi\rangle, |\psi\rangle \in X$, we have $|\varphi\rangle \in \mathcal{R}_{\mathcal{G}_X}(\psi)$ and $|\psi\rangle \in \mathcal{R}_{\mathcal{G}_X}(\varphi)$, where quantum Markov chain $\mathcal{G}_X = \langle X, \mathcal{E}_X \rangle$ is the restriction of $\mathcal{G}$ on X.*

We write $SC(\mathcal{G})$ for the set of strongly connected subspaces of $\mathcal{H}$ in $\mathcal{G}$. It is easy to see that $(SC(\mathcal{G}), \subseteq)$ is an inductive set; that is, for any subset $\{X_i\}$ of $SC(\mathcal{G})$ that is linearly ordered by $\subseteq$, we have $\bigcup_i X_i \in SC(\mathcal{G})$. Thus, by Zorn lemma we assert that there exists a maximal element in $SC(\mathcal{G})$.

Definition 6. *A maximal element of $(SC(\mathcal{G}), \subseteq)$ is called a strongly connected component (SCC) of $\mathcal{G}$.*

To define bottom strongly connected component, we need an auxiliary notion of invariant subspace.

Definition 7. *Let $\mathcal{G} = \langle \mathcal{H}, \mathcal{E} \rangle$ be a quantum Markov chain. Then a subspace X of $\mathcal{H}$ is said to be invariant in $\mathcal{G}$ if $\mathcal{E}(X) \subseteq X$.*

It is easy to see that if super-operator $\mathcal{E}$ has the Kraus representation $\mathcal{E} = \sum_i E_i \cdot E_i^\dagger$, then X is invariant if and only if $E_i X \subseteq X$ for all i. Recall that in a classical Markov chain, the probability of staying in an invariant subset is non-decreasing. A quantum generalisation of this fact is presented in the following:

Theorem 1. *For any invariant subspace X of $\mathcal{H}$ in a quantum Markov chain $\mathcal{G} = \langle \mathcal{H}, \mathcal{E} \rangle$, we have*

$$tr(P_X \mathcal{E}(\rho)) \geq tr(P_X \rho)$$

for all $\rho \in \mathcal{D}(\mathcal{H})$, where P_X is the projection onto X.

Now we are ready to introduce the key notion of this section.

Definition 8. *Let $\mathcal{G} = \langle \mathcal{H}, \mathcal{E} \rangle$ be a quantum Markov chain. Then a subspace X of $\mathcal{H}$ is called a bottom strongly connected component (BSCC) of $\mathcal{G}$ if it is a SCC of $\mathcal{G}$ and invariant in $\mathcal{G}$.*

Example 1. Consider quantum Markov chain $\mathcal{G} = \langle \mathcal{H}, \mathcal{E} \rangle$ with state space $\mathcal{H} = \text{span}\{|0\rangle, \cdots, |4\rangle\}$ and super-operator

$$\mathcal{E} = \sum_{i=1}^{5} E_i \cdot E_i^\dagger,$$

where the operators E_i (i=1,...,5) are given as follows:

$$E_1 = \frac{1}{\sqrt{2}}(|1\rangle\langle 0+1| + |3\rangle\langle 2+3|), \quad E_2 = \frac{1}{\sqrt{2}}(|1\rangle\langle 0-1| + |3\rangle\langle 2-3|),$$
$$E_3 = \frac{1}{\sqrt{2}}(|0\rangle\langle 0+1| + |2\rangle\langle 2+3|), \quad E_4 = \frac{1}{\sqrt{2}}(|0\rangle\langle 0-1| + |2\rangle\langle 2-3|),$$
$$E_5 = \frac{1}{10}(|0\rangle\langle 4| + |1\rangle\langle 4| + |2\rangle\langle 4| + 4|3\rangle\langle 4| + 9|4\rangle\langle 4|),$$

and the states used above are defined by

$$|0 \pm 1\rangle = (|0\rangle \pm |1\rangle)/\sqrt{2} \text{ and } |2 \pm 3\rangle = (|2\rangle \pm |3\rangle)/\sqrt{2}.$$

It is easy to see that $B = \text{span}\{|0\rangle, |1\rangle\}$ is a BSCC of quantum Markov chain $\mathcal{G}$, as for any $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \in B$, we have $\mathcal{E}(\psi) = (|0\rangle\langle 0| + |1\rangle\langle 1|)/2$.

The following lemma clarifies the relationship between different BSCCs.

Lemma 2. *1. For any two different BSCCs X and Y of quantum Markov chain $\mathcal{G}$, we have $X \cap Y = \{0\}$ (0-dimensional Hilbert space).*
2. If X and Y are two BSCCs of $\mathcal{G}$ with $\dim X \neq \dim Y$, then $X \perp Y$.

3.2 Characterisations of BSCCs

This subsection purports to give two characterisations of BSCCs. The first is presented in terms of reachable spaces.

Lemma 3. *A subspace X is a BSCC of quantum Markov chain $\mathcal{G}$ if and only if $\mathcal{R}_{\mathcal{G}}(\phi) = X$ for any non-zero $|\phi\rangle \in X$.*

To present the second characterisation, we need the notion of fixed point of super-operator.

Definition 9. *1. A nonzero partial density operator $\rho \in \mathcal{D}(\mathcal{H})$ is called a fixed point state of super-operator $\mathcal{E}$ if $\mathcal{E}(\rho) = \rho$.*
2. A fixed point state ρ of super-operator $\mathcal{E}$ is called minimal if for any fixed point state σ of $\mathcal{E}$, it holds that $supp(\sigma) \subseteq supp(\rho)$ implies $\sigma = \rho$.

The second characterisation of BSCCs establishes a connection between BSCCs and minimal fixed point states.

Theorem 2. *A subspace X is a BSCC of quantum Markov chain $\mathcal{G} = \langle \mathcal{H}, \mathcal{E} \rangle$ if and only if there exists a minimal fixed point state ρ of $\mathcal{E}$ such that $supp(\rho) = X$. Furthermore, ρ is actually the* unique *fixed point state, up to normalisation, with the support included in X.*

3.3 Checking BSCCs

We now present an algorithm that decides whether or not a given subspace is a BSCC of a quantum Markov chain (see Algorithm 1). The correctness and complexity of this algorithm are given in the following theorem.

Theorem 3. *Given a quantum Markov chain $\langle \mathcal{H}, \mathcal{E} \rangle$ and a subspace $X \subseteq \mathcal{H}$, Algorithm 1 decides whether or not X is a BSCC of $\mathcal{G}$ in time $O(n^6)$, where $n = \dim(\mathcal{H})$.*

4 Decompositions of the State Space

A state in a classical Markov chain is transient if there is a non-zero probability that the process will never return to it, and a state is recurrent if from it the returning probability is 1. It is well-known that a state is recurrent if and only if it belongs to some BSCC in a finite-state Markov chain, and thus the state space of a classical Markov chain can be decomposed into the union of some BSCCs and a transient subspace [2], [16]. The aim of this section is to prove a quantum generalisation of this result.

Definition 10. *A subspace $X \subseteq \mathcal{H}$ is transient in a quantum Markov chain $\mathcal{G} = \langle \mathcal{H}, \mathcal{E} \rangle$ if*

$$\lim_{k \to \infty} \mathrm{tr}(P_X \mathcal{E}^k(\rho)) = 0$$

for any $\rho \in \mathcal{D}(\mathcal{H})$, where P_X is the projection onto X.

The above definition is stated in a "double negation" way. Intuitively, it means that the probability in a transient subspace will be eventually zero. To understand this definition better, let us recall that in a classical Markov chain, a state s is said to be transient if the system starting from s will eventually return to s with probability less than 1. It is well-known that in a finite-state Markov chain, this is equivalent to that the probability at this state will eventually become 0. In the quantum case, the property "eventually return" can be hardly described without measurements, and measurements will disturb the behaviour of the systems. So, we choose to adopt the above definition.

Algorithm 1. CheckBSCC(X)

```
input : A quantum Markov chain G = ⟨H, E⟩ and a subspace X ⊆ H
output: True or False indicating whether X is a BSCC of G
begin
    if E(X) ⋤ X then
        return False;
    end
    E' ← P_X ∘ E;
    B ← a density operator basis of the set {A ∈ B(H) : E'(A) = A};        (*)
    if |B| > 1 then
        return False;
    else
        ρ ← the unique element in B;
        if X = supp(ρ) then
            return True;
        else
            return False;
        end
    end
end
```

To give a characterisation of transient subspaces, we need the notion of the asymptotic average of a super-operator $\mathcal{E}$, which is defined to be

$$\mathcal{E}_\infty = \lim_{N\to\infty} \frac{1}{N} \sum_{n=1}^{N} \mathcal{E}^n. \tag{2}$$

It is easy to see from [21, Proposition 6.3, Proposition 6.9] that $\mathcal{E}_\infty$ is a super-operator as well.

Theorem 4. *The ortho-complement of the image of the state space $\mathcal{H}$ of a quantum Markov chain $\mathcal{G} = \langle \mathcal{H}, \mathcal{E} \rangle$ under the asymptotic average of super-operator $\mathcal{E}$:*

$$T_\mathcal{E} := \mathcal{E}_\infty(\mathcal{H})^\perp$$

is the largest transient subspace in $\mathcal{G}$; that is, any transient subspace of $\mathcal{G}$ is a subspace of $T_\mathcal{E}$.

We now turn to examine the structure of the image of the state space $\mathcal{H}$ under super-operator $\mathcal{E}$.

Theorem 5. *Let $\mathcal{G} = \langle \mathcal{H}, \mathcal{E} \rangle$ be a quantum Markov chain. Then $\mathcal{E}_\infty(\mathcal{H})$ can be decomposed into the direct sum of some orthogonal BSCCs of $\mathcal{G}$.*

Combining Theorems 4 and 5, we see that the state space of a quantum Markov chain $\mathcal{G} = \langle \mathcal{H}, \mathcal{E} \rangle$ can be decomposed into the direct sum of a transient subspace of a family of BSCCs:

$$\mathcal{H} = B_1 \oplus \cdots \oplus B_u \oplus T_\mathcal{E} \tag{3}$$

where B_i's are orthogonal BSCCs of $\mathcal{G}$. A similar decomposition was recently obtained in [18] for a special case of $\mathcal{E}^2 = \mathcal{E}$. The above decomposition holds for any super-operator $\mathcal{E}$ and thus considerably generalises the corresponding result in [18].

The BSCC and transient subspace decomposition of a classical Markov chain is unique. However, it is not the case for quantum Markov chains; a trivial example is that $\mathcal{E}$ is the identity operator, for which any 1-dimensional subspace of $\mathcal{H}$ is a BSCC, and thus for each orthonormal basis $\{|i\rangle\}$ of $\mathcal{H}$, $\bigoplus_i \mathrm{span}\{|i\rangle\}$ is an orthogonal decomposition of $\mathcal{H}$. The following is a more interesting example.

Example 2. Let quantum Markov chain $\mathcal{G} = \langle \mathcal{E}, \mathcal{H} \rangle$ be given as in Example 1. Then $B_1 = \mathrm{span}\{|0\rangle, |1\rangle\}$, $B_2 = \mathrm{span}\{|2\rangle, |3\rangle\}$, $D_1 = \mathrm{span}\{|0+2\rangle, |1+3\rangle\}$, and $D_2 = \mathrm{span}\{|0-2\rangle, |1-3\rangle\}$ are BSCCs, and $T_{\mathcal{E}} = \mathrm{span}\{|4\rangle\}$ is a transient subspace. Furthermore, we have $\mathcal{H} = B_1 \oplus B_2 \oplus T_{\mathcal{E}} = D_1 \oplus D_2 \oplus T_{\mathcal{E}}$.

The relation between different decompositions of a quantum Markov chain is clarified by the following theorem.

Theorem 6. *Let $\mathcal{G} = \langle \mathcal{H}, \mathcal{E} \rangle$ be a quantum Markov chain, and let*

$$\mathcal{H} = B_1 \oplus \cdots \oplus B_u \oplus T_{\mathcal{E}} = D_1 \oplus \cdots \oplus D_v \oplus T_{\mathcal{E}}$$

be two decompositions in the form of Eq. (3), and B_i's and D_i's are arranged, respectively, according to the increasing order of the dimensions. Then $u = v$, and $\dim(B_i) = \dim(D_i)$ for each $1 \leq i \leq u$.

To conclude this section, we present an algorithm for finding a BSCC and transient subspace decomposition of a quantum Markov chain (see Algorithm 2).

Theorem 7. *Given a quantum Markov chain $\langle \mathcal{H}, \mathcal{E} \rangle$, Algorithm 2 decomposes the Hilbert space $\mathcal{H}$ into the direct sum of a family of orthogonal BSCCs and a transient subspace of $\mathcal{G}$ in time $O(n^8)$, where $n = \dim(\mathcal{H})$.*

5 Reachability Probabilities

The traditional way to define reachability probabilities in classical Markov chains is first introducing a probability measure based on cylinder sets of finite paths of states. The probability of reaching a set T is then the probability measure of the set of paths which include a state from T. Typically, reachability probabilities can be obtained by solving a system of linear equations, which is easy and numerically efficient. In quantum Markov chains, however, it is even not clear how to define such a probability measure. Thus it seems hopeless to extend reachability analysis to the quantum case in this way.

Fortunately, there is another way to compute the reachability probability in a classical Markov chain $\langle S, P \rangle$. Given a set of states $T \subseteq S$, we first change the original Markov chain into a new one $\langle S, P' \rangle$ by making states in T absorbing. Then the reachability probability of T is simply the limit of the probability accumulated in T, when the time goes to infinity. It turns out that this equivalent definition can be extended into the quantum case as follows.

Algorithm 2. DecomposeH($\mathcal{G}$)

input : A quantum Markov chain $\mathcal{G} = \langle \mathcal{H}, \mathcal{E} \rangle$
output: A set of orthogonal BSCCs $\{B_i\}$ and a transient subspace $T_{\mathcal{E}}$ such that $\mathcal{H} = \bigoplus_i B_i \oplus T_{\mathcal{E}}$
begin
  $\mathcal{B} \leftarrow$ Decompose($\mathcal{E}_\infty(\mathcal{H})$);
  return $\mathcal{B}$, $\mathcal{E}_\infty(\mathcal{H})^\perp$;
end

Definition 11. *Let $\langle \mathcal{H}, \mathcal{E} \rangle$ be a quantum Markov chain, $\rho \in \mathcal{D}(\mathcal{H})$ an initial state, and $G \subseteq \mathcal{H}$ a subspace. Then the probability of reaching G, starting from ρ, can be defined as*

$$\Pr(\rho \vDash \Diamond G) = \lim_{i \to \infty} \mathrm{tr}(P_G \widetilde{\mathcal{E}}^i(\rho))$$

where $\widetilde{\mathcal{E}} = P_G + \mathcal{E} \circ (I - P_G)$ is the super-operator which first performs the projective measurement $\{P_G, I - P_G\}$ and then applies the identity operator $\mathcal{I}$ or $\mathcal{E}$ depending on the measurement outcome.

Obviously the limit in the above definition exists, as the probabilities $\mathrm{tr}(P_G \widetilde{\mathcal{E}}^i(\rho))$ are nondecreasing in i.

To compute the reachability probability, we first note the subspace G is invariant under $\widetilde{\mathcal{E}}$. Thus $\langle G, \widetilde{\mathcal{E}} \rangle$ is again a quantum Markov chain. Since $\widetilde{\mathcal{E}}(I_G) = I_G$ and $\widetilde{\mathcal{E}}_\infty(G) = G$, we can decompose G into a set of orthogonal BSCCs according to $\widetilde{\mathcal{E}}$ by Theorem 5. The following lemma shows a connection between the limit probability of hitting a BSCC and the probability that the asymptotic average of the initial state lies in the same BSCC.

Lemma 4. *Let $\{B_i\}$ be a BSCC decomposition of $\mathcal{E}_\infty(\mathcal{H})$, and P_{B_i} the projection onto B_i. Then for each i, we have*

$$\lim_{k \to \infty} \mathrm{tr}(P_{B_i} \mathcal{E}^k(\rho)) = \mathrm{tr}(P_{B_i} \mathcal{E}_\infty(\rho)) \tag{4}$$

for all $\rho \in \mathcal{D}(\mathcal{H})$.

Lemma 4 and Theorem 5 together give us an efficient way to compute the reachability probability from a quantum state to a subspace.

Theorem 8. *Let $\langle \mathcal{H}, \mathcal{E} \rangle$ be a quantum Markov chain, $\rho \in \mathcal{D}(\mathcal{H})$, and $G \subseteq \mathcal{H}$ a subspace. Then*

$$\Pr(\rho \vDash \Diamond G) = \mathrm{tr}(P_G \widetilde{\mathcal{E}}_\infty(\rho)),$$

and this probability can be computed in time $O(n^8)$ where $n = \dim(\mathcal{H})$.

Our next results assert that if a quantum Markov chain starts from a pure state in a BSCC then its evolution sequence $\psi, \mathcal{E}(\psi), \mathcal{E}^2(\psi), \cdots$ will hit a subspace with non-zero probability infinitely often provided X is not orthogonal to that BSCC. They establishes

Procedure Decompose(X)

input : A subspace X which is the support of a fixed point state of $\mathcal{E}$
output: A set of orthogonal BSCCs $\{B_i\}$ such that $X = \oplus B_i$
begin
  $\mathcal{E}' \leftarrow P_X \circ \mathcal{E}$;
  $\mathcal{B} \leftarrow$ a density operator basis of the set $\{A \in \mathcal{B}(\mathcal{H}) : \mathcal{E}'(A) = A\}$;
  if $|\mathcal{B}| = 1$ **then**
    $\rho \leftarrow$ the unique element of $\mathcal{B}$;
    return $\{\mathrm{supp}(\rho)\}$;
  else
    $\rho_1, \rho_2 \leftarrow$ two arbitrary elements of $\mathcal{B}$;
    $\rho \leftarrow$ positive part of $\rho_1 - \rho_2$;
    $Y \leftarrow \mathrm{supp}(\rho)^{\perp}$; (* the ortho-complement of $\mathrm{supp}(\rho)$ in X*)
    return $\mathrm{Decompose}(\mathrm{supp}(\rho)) \cup \mathrm{Decompose}(Y)$;
  end
end

indeed a certain fairness and thus can be seen as quantum generalisations of Theorems 10.25 and 10.27 in [2]. It is well-known that in the quantum world a measurement will change the state of the measured system. Consequently, fairness naturally splits into two different versions in quantum Markov chains.

Lemma 5. *(Measure-once fairness) Let B be a BSCC of quantum Markov chain $\mathcal{G} = \langle \mathcal{H}, \mathcal{E} \rangle$, and X a subspace which is not orthogonal to B. Then for any $|\psi\rangle \in B$, it holds that $\mathrm{tr}(P_X \mathcal{E}^i(\psi)) > 0$ for infinitely many i.*

Lemma 6. *(Measure-many fairness) Let B be a BSCC of a quantum Markov chain $\mathcal{G} = \langle \mathcal{H}, \mathcal{E} \rangle$, and $X \subseteq B$ a subspace of B. Then for any $|\psi\rangle \in B$, we have $\lim_{i \to \infty} \mathrm{tr}(\widetilde{\mathcal{E}}^i(\psi)) = 0$, where $\widetilde{\mathcal{E}} = P_{X^\perp} \circ \mathcal{E}$, and $X^\perp$ is the ortho-complement of X in $\mathcal{H}$.*

Lemma 6 is stated also in a "double negation" way. To best understand it, let us assume that at each step after $\mathcal{E}$ is applied, we perform a projective measurement $\{P_X, P_{X^\perp}\}$. If the outcome corresponding to P_X is observed, the process terminates immediately; otherwise, it continues with another round of applying $\mathcal{E}$. Lemma 6 asserts that the probability of nontermination is asymptotically 0; in other words, if we set X as an absorbing boundary, which is included in BSCC B, the reachability probability will be absorbed eventually. This lemma is indeed a strong version of fairness. Furthermore, we have:

Theorem 9. *Let $\mathcal{G} = \langle \mathcal{H}, \mathcal{E} \rangle$ be a quantum Markov chain, and let X be a subspace of $\mathcal{H}$, and $\widetilde{\mathcal{E}} = P_{X^\perp} \circ \mathcal{E}$. Then the following two statements are equivalent:*

1. *The subspace $X^\perp$ contains no BSCC;*
2. *For any $\rho \in \mathcal{D}(\mathcal{H})$, we have $\lim_{i \to \infty} \mathrm{tr}(\widetilde{\mathcal{E}}^i(\rho)) = 0$.*

It is worth noting that in Theorem 9, X is not required to be a subspace of a BSCC B. The following two examples give some simple applications of Theorem 9.

Example 3. Consider a quantum walk on an n-size cycle [1]. The state space of the whole system is $\mathcal{H} = \mathcal{H}_p \otimes \mathcal{H}_c$, where $\mathcal{H}_p = \mathrm{span}\{|0\rangle, \cdots, |n-1\rangle\}$ is the position space, and $\mathcal{H}_c = \mathrm{span}\{|0\rangle, |1\rangle\}$ is the coin space. The evolution of the systems is described by a unitary transformation $U = S(I \otimes H)$, where the coin operator H is the Hadamard operator, and the shift operator

$$S = \sum_{i=0}^{n-1} (|i+1\rangle\langle i| \otimes |0\rangle\langle 0| + |i-1\rangle\langle i| \otimes |1\rangle\langle 1|)$$

where the arithmetic operations over the index set are understood as modulo n. If we set absorbing boundaries at position 0, then from any initial state $|\psi\rangle$, we know from Theorem 9 that the probability of nontermination is asymptotically 0 because there is no BSCC which is orthogonal to the absorbing boundaries.

Example 4. Consider the quantum Markov chain in Example 1. Let ρ_0 be the initial state, and assume that projective measurement $\{P_0 = |0\rangle\langle 0|, P_1 = I - P_0\}$ will be performed at the end of each step and P_0 is set as the absorbing boundary. We write $\tilde{\rho}_k = \tilde{\mathcal{E}}^k(\rho_0)$ for the partial density operator after k steps, where $\tilde{\mathcal{E}} = P_1 \circ \mathcal{E}$.

1. If $\rho_0 = |1\rangle\langle 1|$, then $\lim_{k\to\infty} \tilde{\rho}_k = 0$. This means the probability will be eventually absorbed.
2. If $\rho_0 = |2\rangle\langle 2|$, then $\lim_{k\to\infty} \tilde{\rho}_k = (|2\rangle\langle 2| + |3\rangle\langle 3|)/2$. No probability is absorbed. Let D_1 and D_2 be as in Example 2. Then the probabilities in D_1 and D_2 are both 0.5. This means that if $\mathrm{supp}(P_0)$ is not totally in a BSCC D, then the probability in D may not be absorbed.

6 Repeated Reachability and Persistence Probabilities

In this section, we consider how to compute two kinds of reachability probabilities, namely "repeated reachability" and "persistence property", in a quantum Markov chain. Note that $\mathcal{E}_\infty(\mathcal{H})^\perp$ is a transient subspace. We can focus our attention on $\mathcal{E}_\infty(\mathcal{H})$.

Definition 12. *Let $\mathcal{G} = \langle \mathcal{H}, \mathcal{E} \rangle$ be a quantum Markov chain and G a subspace of $\mathcal{E}_\infty(\mathcal{H})$.*

1. *The set of states in $\mathcal{E}_\infty(\mathcal{H})$ satisfying the repeated reachability* "infinitely often *reaching G" is*

$$\mathcal{X}(G) = \{|\psi\rangle \in \mathcal{E}_\infty(\mathcal{H}) : \lim_{k\to\infty} \mathrm{tr}((P_{G^\perp} \circ \mathcal{E})^k(\psi)) = 0\}.$$

2. *The set of states in $\mathcal{E}_\infty(\mathcal{H})$ satisfying the persistence property* "eventually always *in X" is*

$$\mathcal{Y}(G) = \{|\psi\rangle \in \mathcal{E}_\infty(\mathcal{H}) : (\exists N \geq 0)(\forall k \geq N)\ \mathrm{supp}(\mathcal{E}^k(\psi)) \subseteq G\}.$$

The set $\mathcal{X}(G)$ is defined again in a "double negation" way. Its intuitive meaning can be understood as follows: if the process starts in a state in $\mathcal{X}(G)$ and we make G absorbing, then the probability will be eventually absorbed by G.

The following theorem gives a characterisation of $\mathcal{X}(G)$ and $\mathcal{Y}(G)$ and also clarifies the relationship between them.

Theorem 10. *For any subspace G of $\mathcal{E}_\infty(\mathcal{H})$, both $\mathcal{X}(G)$ and $\mathcal{Y}(G)$ are subspaces of $\mathcal{H}$. Furthermore, we have*

$$\mathcal{X}(G) = \mathcal{E}_\infty(G), \qquad \mathcal{Y}(G) = \bigvee_{B \subseteq G} B = \mathcal{X}(G^\perp)^\perp,$$

where B ranges over all BSCCs, and the orthogonal complements are taken in $\mathcal{E}_\infty(\mathcal{H})$. Moreover, both $\mathcal{X}(G)$ and $\mathcal{Y}(G)$ are invariant.

Example 5. Let us revisit Example 1 where $\mathcal{E}_\infty(\mathcal{H}) = \mathrm{span}\{|0\rangle, |1\rangle, |2\rangle, |3\rangle\}$.

1. If $G = \mathrm{span}\{|0\rangle, |1\rangle, |2\rangle\}$, then $\mathcal{E}_\infty(G^\perp) = \mathrm{supp}(\mathcal{E}_\infty(|3\rangle\langle 3|)) = \mathrm{supp}((|2\rangle\langle 2| + |3\rangle\langle 3|)/2)$ and $\mathcal{E}_\infty(G) = \mathcal{E}_\infty(\mathcal{H})$. Thus $\mathcal{Y}(G) = B_1$ and $\mathcal{X}(G) = \mathcal{E}_\infty(\mathcal{H})$.
2. If $G = \mathrm{span}\{|3\rangle\}$, then $\mathcal{E}_\infty(G^\perp) = B_1 \oplus B_2$ and $\mathcal{E}_\infty(G) = B_2$. Thus $\mathcal{Y}(G) = \{0\}$ and $\mathcal{X}(G) = B_2$.

Now we can define probabilistic persistence and probabilistic repeated reachability.

Definition 13. *1. The probability that a state ρ satisfies the repeated reachability rep(G) is the eventual probability in $\mathcal{X}(G)$, starting from ρ:*

$$\Pr(\rho \vDash \mathit{rep}(G)) = \lim_{k\to\infty} \mathrm{tr}(P_{\mathcal{X}(G)}\mathcal{E}^k(\rho)).$$

2. The probability that a state ρ satisfies the persistence property pers(G) is the eventual probability in $\mathcal{Y}(G)$, starting from ρ:

$$\Pr(\rho \vDash \mathit{pers}(G)) = \lim_{k\to\infty} \mathrm{tr}(P_{\mathcal{Y}(G)}\mathcal{E}^k(\rho)).$$

The well-definedness of the above definition comes from the fact that $\mathcal{X}(G)$ and $\mathcal{Y}(G)$ are invariant. By Theorem 1 we know that the two sequences $\{\mathrm{tr}(P_{\mathcal{X}(G)}\mathcal{E}^k(\rho))\}$ and $\{\mathrm{tr}(P_{\mathcal{Y}(G)}\mathcal{E}^k(\rho))\}$ are non-decreasing, and thus their limits exist. Combining Theorems 4 and 10, we have:

Theorem 11. *1. The repeated reachability probability is*

$$\Pr(\rho \vDash \mathit{rep}(G)) = 1 - \mathrm{tr}(P_{\mathcal{X}(G)^\perp}\mathcal{E}_\infty(\rho)) = 1 - \Pr(\rho \vDash \mathit{pers}(G^\perp)).$$

2. The persistence probability is

$$\Pr(\rho \vDash \mathit{pers}(G)) = \mathrm{tr}(P_{\mathcal{Y}(G)}\mathcal{E}_\infty(\rho)).$$

Finally, we are able to give an algorithm for computing reachability and persistence probabilities (see Algorithm 3).

Theorem 12. *Given a quantum Markov chain $\langle \mathcal{H}, \mathcal{E}\rangle$, an initial state $\rho \in \mathcal{D}(\mathcal{H})$, and a subspace $G \subseteq \mathcal{H}$, Algorithm 3 computes persistence probability $\Pr(\rho \vDash \mathit{pers}(G))$ in time $O(n^8)$, where $n = \dim(\mathcal{H})$.*

With Theorem 11, Algorithm 3 can also be used to compute repeated reachability probability $\Pr(\rho \vDash \mathrm{rep}(G))$.

Algorithm 3. Persistence(G, ρ)

input : A quantum Markov chain $\langle \mathcal{H}, \mathcal{E} \rangle$, a subspace $G \subseteq \mathcal{H}$, and an initial state $\rho \in \mathcal{D}(\mathcal{H})$
output: The probability $\Pr(\rho \models \text{pers}(G))$
begin
  $\rho_\infty \leftarrow \mathcal{E}_\infty(\rho)$;
  $Y \leftarrow \mathcal{E}_\infty(G^\perp)$;
  $P \leftarrow$ the projection onto $Y^\perp$; (* $Y^\perp$ is the ortho-complement of Y in $\mathcal{E}_\infty(\mathcal{H})$ *)
  return $\text{tr}(P\rho_\infty)$;
end

7 Conclusions

We introduced the notion of bottom strongly connected component (BSCC) of a quantum Markov chain (qMC) and studied the BSCC decompositions of qMCs. This enables us to develop an efficient algorithm for computing repeated reachability and persistence probabilities of qMCs. Such an algorithm may be used to verify safety and liveness properties of physical systems produced in quantum engineering and quantum programs for future quantum computers.

Acknowledgments. This work was partially supported by the Australian Research Council (Grant No: DP110103473) and the Overseas Team Program of Academy of Mathematics and Systems Science, Chinese Academy of Sciences.

References

1. Ambainis, A.: Quantum Walks and Their Algorithmic Applications. Int. J. Quantum Inform. 1, 507–518 (2003)
2. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press, Cambridge (2008)
3. Burgarth, D., Chiribella, G., Giovannetti, V., Perinotti, P., Yuasa, K.: Ergodic and Mixing Quantum Channels in Finite Dimensions: arXiv:1210.5625v1
4. Cirac, J.I., Zoller, P.: Goals and Opportunities in Quantum Simulation. Nat. Phys. 8, 264–266 (2012)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, Cambridge (2009)
6. Davidson, T.A.S.: Formal Verification Techniques using Quantum Process Calculus. Ph.D. thesis, University of Warwick (2011)
7. D'Hondt, E., Panangaden, P.: Quantum Weakest Preconditions. Math. Struct. Comp. Sci. 16, 429–451 (2006)
8. Feng, Y., Duan, R.Y., Ying, M.S.: Bisimulation for Quantum Processes. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 523–534. ACM, New York (2011)
9. Feng, Y., Duan, R.Y., Ying, M.S.: Bisimulation for Quantum Processes. ACM T. Progr. Lang. Sys. 34, art. no:17 (2012)

10. Gardiner, C., Zoller, P.: Quantum Noise: A Handbook of Markovian and Non-Markovian Stochastic Methods with Applications to Quantum Optics. Springer, Heidelberg (2004)
11. Gay, S.J., Nagarajan, R.: Communicating Quantum Processes. In: Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (POPL), pp. 145–157. ACM, New York (2005)
12. Gay, S.J., Papanikolaou, N., Nagarajan, R.: Specification and Verification of Quantum Protocols. In: Semantic Techniques in Quantum Computation, pp. 414–472. Cambridge University Press, Cambridge (2010)
13. Gay, S.J., Nagarajan, R., Papanikolaou, N.: QMC: A Model Checker for Quantum Systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 543–547. Springer, Heidelberg (2008)
14. Hart, S., Sharir, M., Pnueli, A.: Termination of Probabilistic Concurrent Programs. ACM T. Progr. Lang. Sys. 5, 356–380 (1983)
15. Jorrand, P., Lalire, M.: Toward a Quantum Process Algebra. In: Proceedings of the First ACM Conference on Computing Frontiers, pp. 111–119. ACM, New York (2004)
16. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomised Algorithms and Probabilistic Analysis. Cambridge University Press, Cambridge (2005)
17. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press, Cambridge (2000)
18. Rosmanis, A.: Fixed Space of Positive Trace-Preserving Super-Operators. Linear Algebra Appl. 437, 1704–1721 (2012)
19. Schirmer, S.G., Solomon, A.I., Leahy, J.V.: Criteria for Reachability of Quantum States. J. Phys. A: Math. Gen. 35, 8551–8562 (2002)
20. Selinger, P.: Towards a Quantum Programming Language. Math. Struct. Comp. Sci. 14, 527–586 (2004)
21. Wolf, M.M.: Quantum Channels and Operators: Guided Tour (unpublished)
22. Yannakakis, M.: Graph-Theoretic Methods in Database Theory. In: Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 230–242. ACM, New York (1990)
23. Ying, M.S.: Floyd-Hoare Logic for Quantum Programs. ACM T. Progr. Lang. Sys. 33, art. no:19 (2011)
24. Yu, N., Ying, M.: Reachability and Termination Analysis of Concurrent Quantum Programs. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 69–83. Springer, Heidelberg (2012)
25. Ying, M.S., Yu, N.K., Feng, Y., Duan, R.Y.: Verification of Quantum Programs. Sci. Comput. Program (accepted, 2013) (also see: arXiv:1106.4063)

Cost Preserving Bisimulations for Probabilistic Automata

Holger Hermanns and Andrea Turrini

Saarland University – Computer Science, Saarbrücken, Germany

Abstract. Probabilistic automata constitute a versatile and elegant model for concurrent probabilistic systems. They are equipped with a compositional theory supporting abstraction, enabled by weak probabilistic bisimulation serving as the reference notion for summarising the effect of abstraction.

This paper considers probabilistic automata augmented with costs. It extends the notions of weak transitions in probabilistic automata in such a way that the costs incurred along a weak transition are captured. This gives rise to cost-preserving and cost-bounding variations of weak probabilistic bisimilarity. Polynomial-time decision algorithms are proposed, that can be effectively used to compute reward-bounding abstractions of Markov decision processes.

1 Introduction

Markov Decision Processes (*MDP*s) are mathematical models widely used in operations research, automated planning, decision support systems and related fields. In the concurrent systems context, they appear in the form of Probabilistic Automata (*PA*s) [18]. *PA*s form the backbone model of successful model checkers such as PRISM [12] enabling the analysis of randomised concurrent systems. They extend classical concurrency models in a simple yet conservative fashion, by enabling probabilistic experiments inside transitions.

As one of the classical concurrency theory manifestations, weak probabilistic bisimilarity is a congruence relation for parallel composition and hiding on *PA*. In other contexts, this has enabled powerful compositional minimisation approaches to combat the state space explosion problem in explicit state verification approaches [6, 10, 15]. With the conception of a polynomial time algorithm for deciding weak probabilistic bisimilarity [11] this avenue can now be followed also in the context of *PA*s and *MDP*s. The decision algorithm follows the usual partition refinement approach. At its core, the decision algorithm needs to check a polynomial number of linear programming (LP) problems. Each of them checks the existence of a specific weak transition. The decision algorithm can be turned into a minimisation algorithm, producing a minimal canonical representation of the *PA* with respect to weak probabilistic bisimilarity [7].

MDP models are usually decorated with cost or rewards structures, with the intention to minimise costs or maximise rewards along the model execution. Likewise, in tools like PRISM, *PA*s appear augmented with cost or reward structures. It is hence a natural question how costs can be embedded into the approach discussed above, and this is what the paper is about.

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 349–363, 2013.

We propose Cost Probabilistic Automata (*CPA*s), a model where *cost* is any kind of quantity associated with the transitions of the automata, and we aim to minimise the cost. For instance, we can consider as the cost of a transition the power needed to transmit a message, the time spent in the computation modelled by the transition, the (monetary) risk associated with an action, the expense of some work, and so on. Costs for weak transitions are interpreted in line with the vast body of literature on *MDP*s, and we describe how that interpretation can be linked to the weak transition encoding as LP problems.

We then extend weak probabilistic bisimulation to also account for costs. As a strict option, we require weak transition costs to be matched exactly for bisimilar states, inducing *cost-preserving weak probabilistic bisimulation*. As a weaker alternative, we ask them to be bounded from one *PA* to the other, leading to the notion of *minor cost weak probabilistic bisimulation*. We provide polynomial time algorithms for both variations. Finally we present an application of minor cost weak probabilistic bisimulation to a multi-hop wireless communication scenario where the cost structure represents transmission power which in turn depends on physical distances.

Organisation of the Paper. After the preliminaries in Section 2, we revisit the LP problem formulation behind weak probabilistic bisimilarity in Section 3 and we present cost probabilistic automata and relative bisimulations in Section 4 together with the wireless channel example. We discuss related work and possible extensions in Section 5 and we conclude the paper in Section 6 with some remarks.

2 Mathematical Preliminaries and Probabilistic Automata

For a set X, denote by $\mathrm{Disc}(X)$ the set of discrete probability distributions over X, and by $\mathrm{SubDisc}(X)$ the set of discrete sub-probability distributions over X. Given $\rho \in \mathrm{SubDisc}(X)$, we denote by $\mathrm{Supp}(\rho)$ the set $\{\, x \in X \mid \rho(x) > 0 \,\}$, by $\rho(\bot)$ the value $1 - \rho(X)$ where $\bot \notin X$, and by δ_x, where $x \in X \cup \{\bot\}$, the *Dirac* distribution such that $\rho(y) = 1$ for $y = x$, 0 otherwise. For a sub-probability distribution ρ, we also write $\rho = \{\, (x, p_x) \mid x \in X \,\}$ where p_x is the probability of x. The lifting $\mathcal{L}(\mathcal{R})$ [14] of a relation $\mathcal{R} \subseteq X \times Y$ is defined as: for $\rho_X \in \mathrm{Disc}(X)$ and $\rho_Y \in \mathrm{Disc}(Y)$, $\rho_X \; \mathcal{L}(\mathcal{R}) \; \rho_Y$ holds if there exists a *weighting function* $w \colon X \times Y \to [0,1]$ such that (1) $w(x,y) > 0$ implies $x \; \mathcal{R} \; y$, (2) $\sum_{y \in Y} w(x,y) = \rho_X(x)$, and (3) $\sum_{x \in X} w(x,y) = \rho_Y(y)$.

A Probabilistic Automaton (*PA*) $\mathcal{A}$ is a tuple $(S, \bar{s}, \Sigma, D)$, where S is a countable set of *states*, $\bar{s} \in S$ is the *start state*, Σ is a countable set of *actions*, and $D \subseteq S \times \Sigma \times \mathrm{Disc}(S)$ is a *probabilistic transition relation*. The set Σ is divided in two sets `H` and `E` of internal (hidden) and external actions, respectively; we let s,t,u,v, and their variants with indices range over S; a, b range over actions; and τ range over internal actions. In this work we consider only finite *PA*s, i.e., *PA*s such that S and D are finite.

A Markov Decision Process (*MDP*) $\mathcal{M}$ is a tuple (S, ι, Σ, P, r) that can be considered as a variation of a *PA* with a functional transition relation $P \colon S \times \Sigma \to \mathrm{Disc}(S)$, a start distribution $\iota \in \mathrm{Disc}(S)$ instead of a start state, and additionally a *reward function* or *structure* $r \colon S \times \Sigma \to \mathbb{R}$. In this paper we consider only non-negative rewards, i.e., $r(s,a) \geq 0$ for each $(s,a) \in S \times \Sigma$, but interpret them as *costs*.

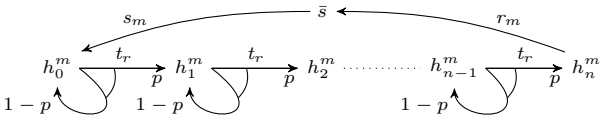


Fig. 1. The wireless communication channel $WCC(n, r, p)$

A transition $tr = (s, a, \mu) \in D$, also denoted by $s \xrightarrow{a} \mu$, is said to *leave* from state s, to be *labelled* by a, and to *lead* to the *target* distribution μ, also denoted by μ_{tr}. We denote by $src(tr)$ the *source* state s and by $act(tr)$ the *action* a. We also say that s enables action a, that action a is enabled from s, and that (s, a, μ) is enabled from s. Finally, we let $D(a) = \{\, tr \in D \mid act(tr) = a \,\}$ be the set of transitions with label a.

Example (A wireless communication channel). As an example of PAs, consider a reliable wireless communication channel used to transmit messages belonging to the set Msg from a sender to a receiver. The wireless implementation of the communication channel is depicted in Fig. 1: the PA $WCC(n, r, p)$ models a communication that requires n intermediate nodes (hops) to reach the receiver where the probability to transmit correctly the message from each node to the successor is p. Each intermediate node has a transmission radius r, and this parameter will become useful when determining the transmission cost in terms of power consumed. In this PA, the message m to transmit is obtained from the sender via the external s_m action and it is delivered to the receiver by using the external action r_m. Internal action t_r models the transmission of the message m from one node to the successor distant at most r, the transmission radius.

The ideal communication channel is modelled by the PA $ICC = WCC(0, \infty, 1)$, that is, the automaton that does not require intermediate nodes. Obviously, ICC models a reliable communication channel since the message is delivered with probability 1 just after having received it.

An *execution fragment* of a PA $\mathcal{A}$ is a finite or infinite sequence of alternating states and actions $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ starting from a state s_0, also denoted by $first(\alpha)$, and, if the sequence is finite, ending with a state denoted by $last(\alpha)$, such that for each $i > 0$ there exists a transition $(s_{i-1}, a_i, \mu_i) \in D$ such that $\mu_i(s_i) > 0$. The *length* of α, denoted by $|\alpha|$, is the number of occurrences of actions in α. If α is infinite, then $|\alpha| = \infty$. Denote by $frags(\mathcal{A})$ the set of execution fragments of $\mathcal{A}$ and by $frags^*(\mathcal{A})$ the set of finite execution fragments of $\mathcal{A}$. An execution fragment α is a *prefix* of an execution fragment α', denoted by $\alpha \leqslant \alpha'$, if the sequence α is a prefix of the sequence α'. The *trace* $trace(\alpha)$ of α is the sub-sequence of external actions of α; we denote by ε the empty trace and we define $trace(a) = a$ for $a \in \mathtt{E}$ and $trace(a) = \varepsilon$ for $a \in \mathtt{H}$.

A *scheduler* for a PA $\mathcal{A}$ is a function $\sigma \colon frags^*(\mathcal{A}) \to \mathrm{SubDisc}(D)$ such that for each $\alpha \in frags^*(\mathcal{A})$, $\sigma(\alpha) \in \mathrm{SubDisc}(\{\, tr \in D \mid src(tr) = last(\alpha) \,\})$. In the MDP context, a scheduler is known as *policy* $\pi \colon frags^*(\mathcal{A}) \to \mathrm{Disc}(D)$. Given a scheduler σ and a finite execution fragment α, the distribution $\sigma(\alpha)$ describes how transitions are chosen to move on from $last(\alpha)$. A scheduler σ and a state s induce a probability distribution $\mu_{\sigma,s}$ over execution fragments as follows. The basic measurable events are the cones of finite execution fragments, where the cone of α, denoted by C_α, is the set

$\{\, \alpha' \in \mathit{frags}(\mathcal{A}) \mid \alpha \leqslant \alpha' \,\}$. The probability $\mu_{\sigma,s}$ of a cone C_α is defined recursively as follows:

$$\mu_{\sigma,s}(C_\alpha) = \begin{cases} 0 & \text{if } \alpha = t \text{ for a state } t \neq s, \\ 1 & \text{if } \alpha = s, \\ \mu_{\sigma,s}(C_{\alpha'}) \cdot \sum_{tr \in D(a)} \sigma(\alpha')(tr) \cdot \mu_{tr}(t) & \text{if } \alpha = \alpha' a t. \end{cases}$$

Standard measure theoretical arguments ensure that $\mu_{\sigma,s}$ extends uniquely to the σ-field generated by cones. We call the resulting measure $\mu_{\sigma,s}$ a *probabilistic execution fragment* of $\mathcal{A}$ and we say that it is generated by σ from s. Given a finite execution fragment α, we define $\mu_{\sigma,s}(\alpha)$ as $\mu_{\sigma,s}(\alpha) = \mu_{\sigma,s}(C_\alpha) \cdot \sigma(\alpha)(\bot)$, where $\sigma(\alpha)(\bot)$ is the probability of terminating the computation after α has occurred.

We say that there is a *weak combined transition* from $s \in S$ to $\mu \in \mathrm{Disc}(S)$ labelled by $a \in \Sigma$, denoted by $s \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu$, if there exists a scheduler σ such that the following holds for the induced probabilistic execution fragment $\mu_{\sigma,s}$: (1) $\mu_{\sigma,s}(\mathit{frags}^*(\mathcal{A})) = 1$; (2) for each $\alpha \in \mathit{frags}^*(\mathcal{A})$, if $\mu_{\sigma,s}(\alpha) > 0$ then $\mathit{trace}(\alpha) = \mathit{trace}(a)$; (3) for each state t, $\mu_{\sigma,s}(\{\, \alpha \in \mathit{frags}^*(\mathcal{A}) \mid \mathit{last}(\alpha) = t \,\}) = \mu(t)$. In this case, we say that the weak combined transition $s \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu$ is induced by σ.

Albeit the definition of weak combined transitions is somewhat intricate, this definition is just the obvious extension of weak transitions on labelled transition systems to the setting with probabilities. See [19] for more details on weak combined transitions.

Example (cont'd). Consider again the *PA* $WCC(2, r, \frac{3}{4})$ and the weak combined transition $h_0^m \stackrel{\tau}{\Longrightarrow}_{\mathrm{C}} \delta_{h_1^m}$. In order to show that it is actually a weak combined transition of the *PA* $WCC(2, r, \frac{3}{4})$, we have to exhibit a scheduler σ inducing it. It is easy to verify that σ defined as: $\sigma(\alpha) = \delta_{h_0^m \xrightarrow{\tau}_\rho}$ if $\mathit{last}(\alpha) = h_0^m$, $\delta_\bot$ otherwise, where $\rho = \{(h_1^m, \frac{3}{4}), (h_0^m, \frac{1}{4})\}$, induces the transition $h_0^m \stackrel{\tau}{\Longrightarrow}_{\mathrm{C}} \delta_{h_1^m}$. Consider, for instance, the probability of stopping in h_1^m, i.e., the sum of the probability of each finite execution fragment ending with h_1^m, i.e., execution fragments of the form $(h_0^m \tau)^{n+1} h_1^m (\tau h_1^m)^l$ where $l, n \in \mathbb{N}$; it is easy to derive that for $n \in \mathbb{N}$, $\mu_{\sigma,h_0^m}((h_0^m \tau)^{n+1} h_1^m) = (\frac{1}{4})^n \cdot \frac{3}{4} \cdot 1 = (\frac{1}{4})^n \cdot \frac{3}{4}$ and that for $l, n \in \mathbb{N}$, $\mu_{\sigma,h_0^m}((h_0^m \tau)^{n+1} h_1^m (\tau h_1^m)^{l+1}) = 0$. Hence we have that $\mu_{\sigma,h_0^m}(\{\, \alpha \in \mathit{frags}^*(\mathcal{A}) \mid \mathit{last}(\alpha) = h_1^m \,\}) = \mu_{\sigma,h_0^m}(\{\, (h_0^m \tau)^{n+1} h_1^m \mid n \in \mathbb{N} \,\}) + \mu_{\sigma,h_0^m}(\{\, (h_0^m \tau)^{n+1} h_1^m (\tau h_1^m)^{l+1} \mid l, n \in \mathbb{N} \,\}) = \sum_{n \in \mathbb{N}} (\frac{1}{4})^n \cdot \frac{3}{4} + 0 = 1 = \delta_{h_1^m}(h_1^m)$.

Weak probabilistic bisimilarity [18, 19] is of central importance for our considerations.

Definition 1. *Let $\mathcal{A}_1$, $\mathcal{A}_2$ be two PAs. An equivalence relation $\mathcal{R}$ on the disjoint union $S_1 \uplus S_2$ is a* weak probabilistic bisimulation *if, for each pair of states $s, t \in S_1 \uplus S_2$ such that $s\ \mathcal{R}\ t$, if $s \stackrel{a}{\longrightarrow} \mu_s$ for some probability distribution μ_s, then there exists μ_t such that $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t$ and $\mu_s\ \mathcal{L}(\mathcal{R})\ \mu_t$.*

We say that $\mathcal{A}_1$ and $\mathcal{A}_2$ are weak probabilistic bisimilar if there exists a weak probabilistic bisimulation $\mathcal{R}$ on $S_1 \uplus S_2$ such that $\bar{s}_1\ \mathcal{R}\ \bar{s}_2$ and we say that two states s_1 and s_2 are weak probabilistic bisimilar if $s_1\ \mathcal{R}\ s_2$. We denote the coarsest weak probabilistic bisimulation, called weak probabilistic bisimilarity, by $\approx$.

Example (cont'd). Consider any instance $WCC(n, r, p)$ and the ideal communication channel ICC. It is quite easy to verify that $ICC \approx WCC(n, r, p)$ for each $n \in \mathbb{N}$,

$r \in \mathbb{R}^{\geq 0}$, and $p \in (0,1]$, where the relation $\mathcal{R}$ justifying $ICC \approx WCC(n,r,p)$ has for each $m \in Msg$ one class containing all h_i^m states and another class containing start states. This means, by transitivity of $\approx$, that $WCC(n,r,p) \approx WCC(n',r',p')$ for each possible value of $n, n' \in \mathbb{N}$, $r, r' \in \mathbb{R}^{\geq 0}$, and $p, p' \in (0,1]$.

We say that there is a *hyper-transition* from $\rho \in \mathrm{Disc}(S)$ to $\mu \in \mathrm{Disc}(S)$ labelled by $a \in \Sigma$, denoted by $\rho \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu$, if there exists a family of weak combined transitions $\{s \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_s\}_{s \in \mathrm{Supp}(\rho)}$ such that $\mu = \sum_{s \in \mathrm{Supp}(\rho)} \rho(s) \cdot \mu_s$, i.e., for each $t \in S$, $\mu(t) = \sum_{s \in \mathrm{Supp}(\rho)} \rho(s) \cdot \mu_s(t)$. Given $s \xrightarrow{a} \rho$ and $\rho \stackrel{\tau}{\Longrightarrow}_{\mathrm{C}} \mu$, we denote by $s \xrightarrow{a} \rho \stackrel{\tau}{\Longrightarrow}_{\mathrm{C}} \mu$ the weak combined transition $s \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu$ obtained by concatenating $s \xrightarrow{a} \rho$ and $\rho \stackrel{\tau}{\Longrightarrow}_{\mathrm{C}} \mu$ (cf. [16, Prop. 3.6]).

3 Weak Transitions as LP Problems Revisited

This section revisits and extends the idea underlying the equivalence of weak transitions and linear programming problems, as developed in [11]. With some inspiration from network flow problems, we were able to see a transition $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t$ of the *PA* $\mathcal{A}$ as a *flow* where the initial probability mass δ_t flows and splits along internal transitions (and exactly one transition with label a for each stream provided $a \neq \tau$) according to the transition target distributions and the scheduler resolution of the nondeterminism.

The LP problem $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond\, \mathcal{L}(\mathcal{E})\ \mu$, proposed in [11] to verify the existence of the weak combined transition $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t$ such that $\mu\ \mathcal{L}(\mathcal{E})\ \mu_t$, assumes that $\mathcal{E}$ is an equivalence relation on S; we can extend it to any relation $\mathcal{R} \subseteq S \times S$ as follows: checking that there exists μ_t such that $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t$ and $\mu\ \mathcal{L}(\mathcal{R})\ \mu_t$ is equivalent, by properties of $\mathcal{L}(\cdot)$, to find μ_t and μ_t' such that $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t$, $\mu_t\ \mathcal{L}(\mathcal{I})\ \mu_t'$, and $\mu\ \mathcal{L}(\mathcal{R})\ \mu_t'$, where $\mathcal{I}$ is the identity relation on S. Since verifying $\mu\ \mathcal{L}(\mathcal{R})\ \mu_t'$ is itself equivalent [2, Lemma 5.1] to solve a maximum flow problem, such flow problem can be merged with the $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond\, \mathcal{L}(\mathcal{I})\ \mu_t'$ LP problem, thereby abstracting the actual distribution μ_t', so as to extend it to a binary relation $\mathcal{R}$, as we formalise in the sequel.

For a *PA* $\mathcal{A} = (S, \bar{s}, \Sigma, D)$ and $\mathcal{R} \subseteq S \times S$, for $a \in \mathtt{E}$, the network $G(t,a,\mu,\mathcal{R}) = (V,E)$ has the set of vertices $V = \{\triangle, \blacktriangledown\} \cup S \cup S^{tr} \cup S_a \cup S_a^{tr} \cup S_{\mathcal{R}}$ where $S^{tr} = \{\, v^{tr} \mid tr = v \xrightarrow{b} \rho \in D, b \in \{a,\tau\} \,\}$, $S_a = \{\, v_a \mid v \in S \,\}$, $S_a^{tr} = \{\, v_a^{tr} \mid v^{tr} \in S^{tr} \,\}$, and $S_{\mathcal{R}} = \{\, s_{\mathcal{R}} \mid s \in S \,\}$ and the set of arcs $E = \{(\triangle, t)\} \cup \{\, (v_a, u_{\mathcal{R}}), (u_{\mathcal{R}}, \blacktriangledown) \mid u, v \in S, v\ \mathcal{R}\ u \,\} \cup \{\, (v, v^{tr}), (v^{tr}, v'), (v_a, v_a^{tr}), (v_a^{tr}, v_a') \mid tr = v \xrightarrow{\tau} \rho \in D, v' \in \mathrm{Supp}(\rho) \,\} \cup \{\, (v, v_a^{tr}), (v_a^{tr}, v_a') \mid tr = v \xrightarrow{a} \rho \in D, v' \in \mathrm{Supp}(\rho) \,\}$. When $a \in \mathtt{H}$, the definition is similar: $V = \{\triangle, \blacktriangledown\} \cup S \cup S^{tr} \cup S_{\mathcal{R}}$ and $E = \{(\triangle, t)\} \cup \{\, (v, u_{\mathcal{R}}), (u_{\mathcal{R}}, \blacktriangledown) \mid u, v \in S, v\ \mathcal{R}\ u \,\} \cup \{\, (v, v^{tr}), (v^{tr}, v') \mid tr = v \xrightarrow{\tau} \rho \in D, v' \in \mathrm{Supp}(\rho) \,\}$.

As in [11], this network $G(t,a,\mu,\mathcal{R})$ and the associated maximum flow problem can not be used directly to encode a weak combined transition since it is not possible to force the flow to split proportional to the transition probability distributions. Instead an ordinary LP problem can be derived from the network, which is enriched with additional constraints called *balancing factors*. A balancing factor models a probabilistic choice and ensures a balance between flows that leave a vertex so as to respect the probability values in a probabilistic choice, i.e., when leaving a vertex $v \in S^{tr} \cup S_a^{tr}$.

Definition 2 (cf. [11, Def. 6]). *Given a PA $\mathcal{A}$, $\mathcal{R} \subseteq S \times S$, $\mu \in \mathrm{Disc}(S)$, and $t \in S$, for $a \in \mathtt{E}$ we define the $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond \mathcal{L}(\mathcal{R})\ \mu$ LP problem associated to the network graph $(V, E) = G(t, a, \mu, \mathcal{R})$ as follows:*

$$
\begin{array}{ll}
\max \sum_{(x,y)\in E} -f_{x,y} & \\
\textit{under constraints} & \\
f_{u,v} \geq 0 & \textit{for each } (u,v) \in E \\
f_{\triangle,t} = 1 & \\
f_{v_{\mathcal{R}},\blacktriangledown} = \mu(v) & \textit{for each } v \in S_{\mathcal{R}} \\
\sum_{u\in\{\, x \mid (x,v)\in E\,\}} f_{u,v} - \sum_{u\in\{\, y\mid (v,y)\in E\,\}} f_{v,u} = 0 & \textit{for each } v \in V \setminus \{\triangle, \blacktriangledown\} \\
f_{v^{tr},v'} - \rho(v') f_{v,v^{tr}} = 0 & \textit{for each } tr = v \xrightarrow{\tau} \rho \in D \textit{ and } v' \in \mathrm{Supp}(\rho) \\
f_{v_a^{tr},v'_a} - \rho(v') f_{v_a,v_a^{tr}} = 0 & \textit{for each } tr = v \xrightarrow{\tau} \rho \in D \textit{ and } v' \in \mathrm{Supp}(\rho) \\
f_{v_a^{tr},v'_a} - \rho(v') f_{v,v_a^{tr}} = 0 & \textit{for each } tr = v \xrightarrow{a} \rho \in D \textit{ and } v' \in \mathrm{Supp}(\rho)
\end{array}
$$

When $a \in \mathtt{H}$, the LP problem $t \stackrel{\tau}{\Longrightarrow}_{\mathrm{C}} \diamond \mathcal{L}(\mathcal{R})\ \mu$ associated to $G(t, \tau, \mu, \mathcal{R})$ is defined as above without the last two groups of constraints.

The objective function has no impact on the equivalence of $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond \mathcal{L}(\mathcal{R})\ \mu$ and a weak combined transition, since any feasible solution is enough to establish the transition (cf. [11, Thm. 8]). This means that we can use $\min \sum_{(x,y)\in E} f_{x,y}$ as objective function, i.e., a weak transition can also be seen as a minimum cost flow problem plus balancing constraints, so we will in the sequel explore how to use the objective function to compute and minimise the cost of performing a weak combined transition.

4 Cost Probabilistic Automata

As said, in this paper we consider as *cost* any kind of quantity associated with the transitions of the automaton $\mathcal{A}$ that we aim to minimise. We model the cost of the transitions by a function c that assigns to each transition a non-negative real value.

Definition 3. *A* cost probabilistic automaton *(CPA) is a pair $(\mathcal{A}, c)$ where $\mathcal{A}$ is a probabilistic automaton and c, the* transition cost function*, is a total function $c\colon D \to \mathbb{R}^{\geq 0}$.*

4.1 Weak Combined Transition Cost

There are several ways of extending the cost from a single transition to a sequence of transitions, and hence to a weak combined transition. One possibility is to consider the weighted sum of the costs of all involved finite execution fragments. This approach matches the standard interpretation in the operations research literature for expected reward criteria [13].

Definition 4. *Given an MDP $\mathcal{M} = (S, \iota, \Sigma, P, r)$, a finite execution fragment $\alpha = s_1 a_1 \ldots s_n a_n s_{n+1} \in \mathit{frags}^*(\mathcal{M})$, a policy π, and the final state reward $r_s\colon S \to \mathbb{R}$, let $\alpha\lfloor i = s_1 a_1 \ldots a_{i-1} s_i$ be the i-prefix of α, $r(\alpha) = \sum_{i=1}^{n} r(s_i, a_i) + r_s(s_{n+1})$, and $P^\pi(\alpha) = \iota(s_1) \cdot \prod_{i=1}^{n} \pi(\alpha\lfloor i)(a_i) \cdot P(s_i, a_i)(s_{i+1})$. Then the* expected total reward with horizon N *is defined as $\mathbb{E}^\pi_N = \sum_{\alpha\in\{\, \alpha\in \mathit{frags}^*(\mathcal{M}) \mid |\alpha|=N\,\}} r(\alpha) \cdot P^\pi(\alpha)$.*

Since probabilistic automata are a conservative extension of *MDP*, we extend this notion to weak transition costs by taking into account the resolution of the nondeterminism as induced by a given scheduler.

Definition 5. *Given a CPA $(\mathcal{A}, c)$, a state s, an action a, a probability distribution μ, and a scheduler σ inducing the weak combined transition $s \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu$, we define the cost $c_\sigma(s \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu)$ of the weak combined transition $s \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu$ as*

$$c_\sigma(s \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu) = \sum_{\alpha \in frags^*(\mathcal{A})} c_\sigma(\alpha) \cdot \mu_{\sigma,s}(\alpha)$$

where $c_\sigma(\alpha) = c_\sigma(\alpha') + \sum_{tr \in D(a)} c(tr) \cdot \widehat{\sigma}(\alpha', t, a, tr)$ if $\alpha = \alpha' a t$, 0 otherwise, and where $\widehat{\sigma} \colon frags^*(\mathcal{A}) \times S \times \Sigma \times D \to \mathbb{R}^{\geq 0}$ is defined as:

$$\widehat{\sigma}(\alpha, t, a, tr) = \begin{cases} \dfrac{\sigma(\alpha)(tr) \cdot \mu_{tr}(t)}{\sum_{tr \in D(a)} \sigma(\alpha)(tr) \cdot \mu_{tr}(t)} & \text{if } \sum_{tr \in D(a)} \sigma(\alpha)(tr) \cdot \mu_{tr}(t) > 0, \\ 0 & \text{otherwise.} \end{cases}$$

When the scheduler σ is clear from the context, we just write $c(s \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu)$.

When we restrict Def. 5 to *MDP*s, it coincides with Def. 4:

Proposition 1. *Given an MDP $\mathcal{M}$, a policy π, and a final state reward $r \colon S \to \mathbb{R}$ such that for each $s \in S$, $r(s) = 0$, it holds that for each $N \in \mathbb{N}$,*

$$\mathbb{E}^\pi_N = \sum_{s \in S} \iota(s) \cdot c(s \stackrel{\tau}{\Longrightarrow}_{\mathrm{C}} \mu)$$

where for each $tr = (s, a, P(s,a)) \in D$ and $\alpha \in frags^*(\mathcal{M})$, we define $c(tr) = r(s,a)$, $\sigma(\alpha)(tr) = \pi(\alpha)(a)$ if $|\alpha| < N$, 0 otherwise, and $s \stackrel{\tau}{\Longrightarrow}_{\mathrm{C}} \mu$ is the weak combined transition induced by the scheduler σ when all actions are considered as internal.

Example (cont'd). Consider the *CPA* $(WCC(n, r, p), c)$ where c assigns cost $\mathbf{r}$ to each transition labelled by the internal action t_r; the weak combined transition $h_0^m \stackrel{t_r}{\Longrightarrow}_{\mathrm{C}} \delta_{h_n^m}$ can be seen as the sequence of transitions $h_i^m \stackrel{t_r}{\Longrightarrow}_{\mathrm{C}} \delta_{h_{i+1}^m}$ for $0 \leq i < n$. It is routine to check that each $h_i^m \stackrel{t_r}{\Longrightarrow}_{\mathrm{C}} \delta_{h_{i+1}^m}$ is induced by the scheduler σ_i such that $\sigma_i(\alpha) = \delta_{tr_i^m}$ if $last(\alpha) = h_i^m$, $\delta_\perp$ otherwise, where $tr_i^m = h_i^m \xrightarrow{t_r} \{(h_{i+1}^m, p), (h_i^m, 1-p)\}$. Now, consider the finite execution fragment $\alpha = (h_i^m t_r)^{n+1} h_{i+1}^m$: according to Def. 5, it has cost $c_{\sigma_i}(\alpha) = (n+1) \cdot \mathbf{r}$. The probability $\mu_{\sigma_i, h_i^m}(\alpha)$ of α is $(1-p)^n \cdot p$ while the probability of each $\alpha' \in frags^*(WCC(n,r,p)) \setminus \{ (h_i^m t_r)^{n+1} h_{i+1}^m \mid n \in \mathbb{N} \}$ is 0, thus the cost of the transition $h_i^m \stackrel{t_r}{\Longrightarrow}_{\mathrm{C}} \delta_{h_{i+1}^m}$ as induced by σ_i is $c_{\sigma_i}(h_i^m \stackrel{t_r}{\Longrightarrow}_{\mathrm{C}} \delta_{h_{i+1}^m}) = \sum_{n \in \mathbb{N}} (n+1) \cdot \mathbf{r} \cdot (1-p)^n \cdot p = \mathbf{r} \cdot p \cdot \sum_{n \in \mathbb{N}} (n+1) \cdot (1-p)^n = \frac{\mathbf{r} \cdot p}{1-p} \cdot \sum_{n \in \mathbb{N}} (n+1) \cdot (1-p)^{n+1} = \frac{\mathbf{r} \cdot p}{1-p} \cdot \frac{1-p}{(1-(1-p))^2} = \frac{\mathbf{r}}{p}$, hence $h_0^m \stackrel{t_r}{\Longrightarrow}_{\mathrm{C}} \delta_{h_n^m}$ has cost $n \cdot \frac{\mathbf{r}}{p}$.

By using an equivalent definition of weak transition cost, the transition costs can be encoded in the LP problem as coefficients of the objective function.

Definition 6. *Given a CPA $(\mathcal{A}, c)$, a binary relation $\mathcal{R}$ on S, a probability distribution $\mu \in \mathrm{Disc}(S)$, and a state $t \in S$, for action $a \neq \tau$ we define the min-cost LP problem $\min_c t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \circledast\, \mathcal{L}(\mathcal{R})\ \mu$ associated to the network $G(t, a, \mu, \mathcal{R})$ as follows.*

$$
\begin{array}{ll}
\min \sum_{(x,y)\in E} c_f((x,y)) \cdot f_{x,y} & \\
\textit{under constraints} & \\
f_{u,v} \geq 0 & \textit{for each } (u,v) \in E \\
f_{\triangle,t} = 1 & \\
f_{v_{\mathcal{R}},\blacktriangledown} = \mu(v) & \textit{for each } v \in S_{\mathcal{R}} \\
\sum_{u\in\{\, x \mid (x,v)\in E\,\}} f_{u,v} - \sum_{u\in\{\, y \mid (v,y)\in E\,\}} f_{v,u} = 0 & \textit{for each } v \in V \setminus \{\triangle, \blacktriangledown\} \\
f_{v^{tr},v'} - \rho(v') f_{v,v^{tr}} = 0 & \textit{for each } tr = v \xrightarrow{\tau} \rho \in D \textit{ and } v' \in \mathrm{Supp}(\rho) \\
f_{v_a^{tr},v'_a} - \rho(v') f_{v_a,v_a^{tr}} = 0 & \textit{for each } tr = v \xrightarrow{\tau} \rho \in D \textit{ and } v' \in \mathrm{Supp}(\rho) \\
f_{v_a^{tr},v'_a} - \rho(v') f_{v,v_a^{tr}} = 0 & \textit{for each } tr = v \xrightarrow{a} \rho \in D \textit{ and } v' \in \mathrm{Supp}(\rho)
\end{array}
$$

where $c_f \colon E \to \mathbb{R}^{\geq 0}$ *is a total function defined as follows:*

$$
c_f((x,y)) = \begin{cases} c(tr) & \textit{if } tr = v \xrightarrow{\tau} \rho,\ x = v,\ y = v^{tr}, \\ c(tr) & \textit{if } tr = v \xrightarrow{\tau} \rho,\ x = v_a,\ y = v_a^{tr}, \\ c(tr) & \textit{if } tr = v \xrightarrow{a} \rho,\ x = v,\ y = v_a^{tr}, \\ 0 & \textit{otherwise.} \end{cases}
$$

If $\min_c t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond\, \mathcal{L}(\mathcal{R})\ \mu$ *has an optimal solution* f^o*, then we denote by* $\mathfrak{C}$ *the minimum cost* $\mathfrak{C} = \sum_{(x,y)\in E} c_f((x,y)) \cdot f^o_{x,y}$.

When the action a is τ, the min-cost LP problem $\min_c t \stackrel{\tau}{\Longrightarrow}_{\mathrm{C}} \diamond\, \mathcal{L}(\mathcal{R})\ \mu$ associated to the network $G(t,\tau,\mu,\mathcal{R})$ is defined as above without the last two groups of constraints.

A first straightforward result is that $\min_c t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond\, \mathcal{L}(\mathcal{R})\ \mu$ is feasible if and only if $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond\, \mathcal{L}(\mathcal{R})\ \mu$ is feasible, since the only difference between the two problems is the objective function that does not affect the feasibility of an LP problem:

Proposition 2. *Given a CPA* $(\mathcal{A}, c)$*,* $\mathcal{R} \subseteq S \times S$*,* $a \in \Sigma$*,* $\mu \in \mathrm{Disc}(S)$*, and* $t \in S$*, the minimisation LP problem* $\min_c t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond\, \mathcal{L}(\mathcal{R})\ \mu$ *has a feasible solution* f^* *if and only if* f^* *is a feasible solution of the LP problem* $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond\, \mathcal{L}(\mathcal{R})\ \mu$*.*

Similarly, as generating and checking the existence of a valid solution of the LP problem $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond\, \mathcal{L}(\mathcal{R})\ \mu$ is polynomial in $N = \max\{|S|, |D|\}$ (cf. [11, Thm. 7]), the same holds for $\min_c t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond\, \mathcal{L}(\mathcal{R})\ \mu$:

Corollary 1. *Given a CPA* $(\mathcal{A}, c)$*,* $\mathcal{R} \subseteq S \times S$*,* $a \in \Sigma$*,* $\mu \in \mathrm{Disc}(S)$*, and* $t \in S$*, generating and checking the existence of a valid solution of the minimisation LP problem* $\min_c t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond\, \mathcal{L}(\mathcal{R})\ \mu$ *is polynomial in* $N = \max\{|S|, |D|\}$*.*

Since $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond\, \mathcal{L}(\mathcal{R})\ \mu$ is feasible if and only if there exists a scheduler σ that induces $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t$ such that $\mu\ \mathcal{L}(\mathcal{R})\ \mu_t$, we may expect a similar result regarding costs, that is, $\min_c t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond\, \mathcal{L}(\mathcal{R})\ \mu$ is feasible with optimal value $\mathfrak{C}$ if and only if there exists a scheduler σ that induces $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t$ such that $\mu\ \mathcal{L}(\mathcal{R})\ \mu_t$ and $c(t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t) = \mathfrak{C}$. But in general it is not possible to obtain such a result since there can be equivalent ways to resolve nondeterminism that induce different costs, thus we can not talk about *the cost* of a weak combined transition, but of the cost of the weak combined transition *as induced by the scheduler* σ. For instance, consider an automaton $\mathcal{A}$ whose transitions are $tr_1 = \bar{s} \xrightarrow{a} \delta_t$, $tr_2 = \bar{s} \xrightarrow{\tau} \delta_v$, and $tr_3 = v \xrightarrow{a} \delta_t$, each one with cost 1. It is straightforward to check that the scheduler σ_1 such that $\sigma_1(\bar{s}) = \delta_{tr_1}$ and $\sigma_1(\alpha) = \delta_\perp$ for each

finite execution fragment $\alpha \neq \bar{s}$ induces the weak combined transition $\bar{s} \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \delta_t$ whose cost is 1; the same transition is induced also by the scheduler σ_2 defined as $\sigma_2(\bar{s}) = \delta_{tr_2}$, $\sigma_2(\bar{s}\tau v) = \delta_{tr_3}$, and $\sigma_2(\alpha) = \delta_\perp$ for each other finite execution fragment α. However the cost as induced by σ_2 is $c_{\sigma_2}(\bar{s} \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \delta_t) = 2 \neq 1 = c_{\sigma_1}(\bar{s} \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \delta_t)$; it is easy to show that $1 \leq c_\sigma(\bar{s} \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \delta_t) \leq 2$ for each scheduler σ inducing $\bar{s} \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \delta_t$. Note that there are uncountably many such schedulers, each one corresponding to a different resolution of the choice between $tr_1 = \bar{s} \stackrel{a}{\longrightarrow} \delta_t$ and $tr_2 = \bar{s} \stackrel{\tau}{\longrightarrow} \delta_v$: in general, we can denote such choice as the distribution $\{(tr_1, p), (tr_2, 1-p)\}$ where $p \in [0, 1]$.

The cost given by a scheduler and the value of the objective function of the corresponding LP problem are however related:

Theorem 1. *Given a CPA $(\mathcal{A}, c)$, $\mathcal{R} \subseteq S \times S$, $a \in \Sigma$, $\mu \in \mathrm{Disc}(S)$, and $t \in S$, consider the $\min_c t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond \mathcal{L}(\mathcal{R})\ \mu$ LP problem. The following implications hold:*

1. *If there exists a scheduler σ for $\mathcal{A}$ that induces $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t$ such that $\mu\ \mathcal{L}(\mathcal{R})\ \mu_t$, then $\min_c t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond \mathcal{L}(\mathcal{R})\ \mu$ has an optimal solution f^o such that $\mathfrak{C} \leq c(t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t)$.*
2. *If $\min_c t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond \mathcal{L}(\mathcal{R})\ \mu$ has an optimal solution f^o, then there exists a scheduler σ for $\mathcal{A}$ that induces $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t$ such that $\mu\ \mathcal{L}(\mathcal{R})\ \mu_t$ and $c(t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t) = \mathfrak{C}$.*

As immediate corollaries we have that the cost given by the optimal solution of the $\min_c t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond \mathcal{L}(\mathcal{R})\ \mu$ LP problem corresponds to the minimum cost induced by any scheduler inducing $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t$ and that finding such minimum is polynomial.

Corollary 2. *Given a CPA $(\mathcal{A}, c)$, $\mathcal{R} \subseteq S \times S$, $a \in \Sigma$, $\mu \in \mathrm{Disc}(S)$, and $t \in S$ such that there exists $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t$ with $\mu\ \mathcal{L}(\mathcal{R})\ \mu_t$, the LP problem $\min_c t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \diamond \mathcal{L}(\mathcal{R})\ \mu$ has minimum cost $\mathfrak{C} = \min\{\, c_\sigma(t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t) \mid \sigma$ induces $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t$ such that $\mu\ \mathcal{L}(\mathcal{R})\ \mu_t \,\}$.*

Corollary 3. *Given a CPA $(\mathcal{A}, c)$, $\mathcal{R} \subseteq S \times S$, $a \in \Sigma$, $\mu \in \mathrm{Disc}(S)$, and $t \in S$, finding $\min\{\, c_\sigma(t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t) \mid \sigma$ induces $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t$ such that $\mu\ \mathcal{L}(\mathcal{R})\ \mu_t \,\}$ is polynomial in $N = \max\{|S|, |D|\}$.*

Extending the above results to hyper-transitions of the *CPA* $(\mathcal{A}, c)$ is straightforward, since we can consider each hyper-transition $\rho \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu$ as the weak combined transition $h \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu$ in the *CPA* $(\mathcal{A}', c')$ that is $(\mathcal{A}, c)$ enriched with the fresh state h and the transition $h \stackrel{\tau}{\longrightarrow} \rho$ whose cost is set to 0.

4.2 Cost Preserving Bisimulations

We now discuss options for weak bisimulations on *CPA*, so as to ignore internal computations as long as those do not change the visible behaviour of the system. Since a *CPA* is an ordinary *PA* enriched with a cost function, one might consider a naive lifting of *PA* weak probabilistic bisimulation, where two *CPA* are weak probabilistic bisimilar if the underlying *PA* are. However this definition obviously falls too short, since it may relate states with different cost behaviours. For this reason, following [9], we define a restricted notion of weak probabilistic bisimulation where each transition $s \stackrel{a}{\longrightarrow} \mu_s$ of the challenging state s has to be matched by the defender state t by enabling a weak combined transition $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t$ such that $\mu_s\ \mathcal{L}(\mathcal{R})\ \mu_t$ as in ordinary weak probabilistic bisimulation, and, in addition, the costs of challenging and defending transitions must agree.

Definition 7. *Given two CPAs $(\mathcal{A}_1, c_1)$ and $(\mathcal{A}_2, c_2)$, an equivalence relation $\mathcal{R}$ on the disjoint union $S_1 \uplus S_2$ is a* weak probabilistic cost-preserving bisimulation *if for each pair of states $s, t \in S_1 \uplus S_2$ such that $s\ \mathcal{R}\ t$, if $s \xrightarrow{a} \mu_s$, then there exists μ_t such that $t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t$, $\mu_s\ \mathcal{L}(\mathcal{R})\ \mu_t$, and $c_d(t \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_t) = c_c(s \xrightarrow{a} \mu_s)$ where c_d and c_c are the cost functions of the defender and the challenger CPA, respectively.*

Two CPAs $(\mathcal{A}_1, c_1)$ and $(\mathcal{A}_2, c_2)$ are weak probabilistic cost-preserving bisimilar if there exists a weak probabilistic cost-preserving bisimulation $\mathcal{R}$ on $S_1 \uplus S_2$ such that $\bar{s}_1\ \mathcal{R}\ \bar{s}_2$. We denote the coarsest weak probabilistic cost-preserving bisimulation by $\approx_c$, called weak probabilistic cost-preserving bisimilarity.

By using this definition of bisimulation, we have that states enabling transitions with different cost are no more bisimilar, since they do not respect cost constraints.

4.3 Cost Bounding Bisimulations

The definition of weak probabilistic cost-preserving bisimulation allows us to relate CPAs that have the same behaviour and the same cost. Since our aim is to minimise the cost while preserving the behaviour of a CPA $(\mathcal{A}, c)$, we will now relax the cost equality by requiring that the cost of the defender matching transition is at most the cost of the challenger transition. Despite the simplicity of this idea, the formal definition is quite involved since we have to consider properly the cost of internal transitions.

To shed some light on this, consider an automaton $\mathcal{A}_1$ performing three internal steps $\bar{s}_1 \xrightarrow{\tau} \delta_{t_1}$, $t_1 \xrightarrow{\tau} \delta_{u_1}$, and $u_1 \xrightarrow{\tau} \delta_{v_1}$ where each step has cost 5 followed by an external step $v_1 \xrightarrow{a} \delta_{x_1}$ with cost 1 and an automaton $\mathcal{A}_2$ that performs four steps $\bar{s}_2 \xrightarrow{\tau} \delta_{t_2}$, $t_2 \xrightarrow{\tau} \delta_{u_2}$, $u_2 \xrightarrow{\tau} \delta_{v_2}$, and $v_2 \xrightarrow{\tau} \delta_{w_2}$ each with cost 3 followed by an external step $w_2 \xrightarrow{a} \delta_{x_2}$ with cost 1. An external observer is able to recognise that the behaviour of $\mathcal{A}_1$ is more expensive than the one of $\mathcal{A}_2$ since the overall cost is 16 for the former, 13 for the latter. However, from a state-based bisimulation point of view, $\mathcal{A}_2$ is not always cheaper than $\mathcal{A}_1$: let $\{\{\bar{s}_1, \bar{s}_2\}, \{t_1, t_2\}, \{u_1, u_2\}, \{v_1, v_2, w_2\}, \{x_1, x_2\}\}$ be the equivalence classes of $\mathcal{R}$; it is easy to verify that $\mathcal{R}$ is a weak probabilistic bisimulation between $\mathcal{A}_1$ and $\mathcal{A}_2$: when $\mathcal{A}_1$ performs $\bar{s}_1 \xrightarrow{\tau} \delta_{t_1}$ with cost 5, $\mathcal{A}_2$ replies with $\bar{s}_2 \xrightarrow{\tau} \delta_{t_2}$ with cost $3 \leq 5$ and $t_1\ \mathcal{R}\ t_2$. Note that $\mathcal{A}_2$ can not perform the subsequent transition $t_2 \xrightarrow{\tau} \delta_{u_2}$ since in this case the overall cost is $6 \not\leq 5$. The same happens for transitions $t_1 \xrightarrow{\tau} \delta_{u_1}$ and $u_1 \xrightarrow{\tau} \delta_{v_1}$ that are matched by $t_2 \xrightarrow{\tau} \delta_{u_2}$ and $u_2 \xrightarrow{\tau} \delta_{v_2}$, respectively. Since $\mathcal{A}_1$ now performs $v_1 \xrightarrow{a} \delta_{x_1}$ with cost 1, v_2 is not able to match this transition with a cost at most 1: in order to match the transition, $\mathcal{A}_2$ has to perform both transitions $v_2 \xrightarrow{\tau} \delta_{w_2}$ and $w_2 \xrightarrow{a} \delta_{x_2}$ whose cost is $4 \not\leq 1$.

These considerations indicate that internal challenger transitions should not be considered separately but as a whole, so in order to abstract away from costs of single challenger internal transitions while preserving the overall cost, we consider for the challenger the cost of reaching the border states, i.e., states where the automaton performs an external action or exhibits a different behaviour by changing the current class as induced by the weak bisimulation relation.

Definition 8. *Given a PA $\mathcal{A}$ and an equivalence relation $\mathcal{R}$ over S, we say that a state s is a* border state *if there exists $s \xrightarrow{a} \mu \in D$ such that either $\mu([s]_{\mathcal{R}}) < 1$ or $a \in \mathtt{E}$.*

We denote the set of all border states with respect to $\mathcal{R}$ by $\mathcal{B}(\mathcal{R})$.

Definition 9. *Let $(\mathcal{A}_1, c_1)$ and $(\mathcal{A}_2, c_2)$ be two CPAs. Let $\mathcal{W}$ be an equivalence relation on the disjoint union $S_1 \uplus S_2$ and $\mathcal{C} \subseteq \mathcal{W} \cap S_2 \times S_1$ such that for each $s_2 \in S_2$ there exists $s_1 \in S_1$ such that $s_2 \; \mathcal{C} \; s_1$. Then we say that $(\mathcal{W}, \mathcal{C})$ is a* minor cost weak probabilistic bisimulation *from $(\mathcal{A}_1, c_1)$ to $(\mathcal{A}_2, c_2)$ if $\mathcal{W}$ is a weak probabilistic bisimulation for $\mathcal{A}_1$ and $\mathcal{A}_2$ and for each $s_2 \xrightarrow{a} \mu_2 \in D_2$ and each $s_1 \in S_1$ such that $s_2 \; \mathcal{C} \; s_1$,*

1. *if there exists $\rho_2 \in \mathrm{Disc}(\mathcal{B}(\mathcal{W}) \cap S_2)$ such that $\mu_2 \stackrel{\tau}{\Longrightarrow}_{\mathrm{C}} \rho_2$, then there exists $\rho_1 \in \mathrm{Disc}(\mathcal{B}(\mathcal{W}) \cap S_1)$ such that (a) $s_1 \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \rho_1$, (b) $\rho_2 \; \mathcal{L}(\mathcal{C}) \; \rho_1$, (c) $c_1(s_1 \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \rho_1) \leq c_2(s_2 \xrightarrow{a} \mu_2 \stackrel{\tau}{\Longrightarrow}_{\mathrm{C}} \rho_2)$, and (d) $\min\{\, c_2(\mu_2 \stackrel{\tau}{\Longrightarrow}_{\mathrm{C}} \rho) \mid \rho \in \mathrm{Disc}(\mathcal{B}(\mathcal{W}) \cap S_2) \,\} = c_2(\mu_2 \stackrel{\tau}{\Longrightarrow}_{\mathrm{C}} \rho_2)$; or*
2. *if there does not exist $\rho_2 \in \mathrm{Disc}(\mathcal{B}(\mathcal{W}) \cap S_2)$ such that $\mu_2 \stackrel{\tau}{\Longrightarrow}_{\mathrm{C}} \rho_2$, then there exists $\mu_1 \in \mathrm{Disc}(S_1)$ such that $s_1 \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_1$, $\mu_2 \; \mathcal{L}(\mathcal{C}) \; \mu_1$, and $c_1(s_1 \stackrel{a}{\Longrightarrow}_{\mathrm{C}} \mu_1) \leq c_2(s_2 \xrightarrow{a} \mu_2)$.*

We say that $(\mathcal{A}_1, c_1)$ is minor cost weak probabilistic bisimilar to $(\mathcal{A}_2, c_2)$ if there exists a minor cost weak probabilistic bisimulation $(\mathcal{W}, \mathcal{C})$ such that $\bar{s}_2 \; \mathcal{C} \; \bar{s}_1$. We denote the coarsest minor cost weak probabilistic bisimulation from $(\mathcal{A}_1, c_1)$ to $(\mathcal{A}_2, c_2)$ by $(\mathcal{A}_1, c_1) \lessapprox_c (\mathcal{A}_2, c_2)$ and we say that $(\mathcal{A}_1, c_1)$ is in minor cost weak probabilistic bisimilarity with $(\mathcal{A}_2, c_2)$.

4.4 The Cost of the Wireless Communication Channel

We now apply the minor cost weak probabilistic bisimulation to the reliable wireless communication channel introduced in Sec. 2 and depicted in Fig. 1. As cost, we consider the function c that assigns cost 1 to transitions labelled by s_m or r_m and cost r^2 to transitions labelled by t_r. We use value 1 to represent a constant power consumption relative to sending/receiving message actions and value r^2 to model the energy, quadratic on the transmission radius, required to transmit a message via wireless.

As a concrete example, consider the two instances $\mathcal{A}_{23} = WCC(2, 3, \frac{1}{2})$ and $\mathcal{A}_{32} = WCC(3, 2, \frac{1}{2})$ of the wireless communication channel connecting sender and receiver that are at distance 6. To avoid name collisions, we rename the states h_j^m of $WCC(3, 2, \frac{1}{2})$ to k_j^m for $0 \leq j \leq 3$. It is easy to verify that the equivalence relation $\mathcal{W}$ whose classes are $\{\bar{s}_{23}, \bar{s}_{32}\}$ and $\{\, h_i^m, k_j^m \mid 0 \leq i \leq 2, 0 \leq j \leq 3 \,\}$ for each $m \in Msg$ justifies $\mathcal{A}_{23} \approx \mathcal{A}_{32}$, so consider the two CPAs $(\mathcal{A}_{23}, c)$ and $(\mathcal{A}_{32}, c)$. We suspect that $(\mathcal{A}_{32}, c) \lessapprox_c (\mathcal{A}_{23}, c)$, but not the reverse, since intuitively $(\mathcal{A}_{23}, c)$ has overall cost 26 for sending and receiving a single message while $(\mathcal{A}_{32}, c)$ has overall cost 38. In order to show $(\mathcal{A}_{32}, c) \lessapprox_c (\mathcal{A}_{23}, c)$, we have to find a suitable relation $\mathcal{C}$ that, together with $\mathcal{W}$, satisfies the conditions of Def. 9. A suitable relation is $\mathcal{C} = \{(\bar{s}_{23}, \bar{s}_{32})\} \cup \bigcup_{m \in Msg} \{\, (h_i^m, k_3^m) \mid 0 \leq i \leq 2 \,\}$: consider the pair $(\bar{s}_{23}, \bar{s}_{32})$ and the only available transition $\bar{s}_{23} \xrightarrow{s_m} \delta_{h_0^m}$. Since $\mathcal{B}(\mathcal{W}) = \{\, \bar{s}_{23}, \bar{s}_{32}, h_2^m, k_3^m \mid m \in Msg \,\}$, the only possible $\rho_{23} \in \mathrm{Disc}(\mathcal{B}(\mathcal{W}) \cap S_{23})$ such that $\delta_{h_0^m} \stackrel{\tau}{\Longrightarrow} \rho_{23}$ is $\rho_{23} = \delta_{h_2^m}$. In order to match such transition, $\bar{s}_{32}$ enables the weak transition $\bar{s}_{32} \stackrel{s_m}{\Longrightarrow}_{\mathrm{C}} \delta_{k_3^m}$ that satisfies $\delta_{h_2^m} \; \mathcal{L}(\mathcal{C}) \; \delta_{k_3^m}$. The last condition we have to verify is that $c(\bar{s}_{32} \stackrel{s_m}{\Longrightarrow}_{\mathrm{C}} \delta_{k_3^m}) \leq c(\bar{s}_{23} \xrightarrow{s_m} \delta_{h_0^m} \stackrel{\tau}{\Longrightarrow}_{\mathrm{C}} \delta_{h_2^m})$; this constraint is satisfied since $c(\bar{s}_{32} \stackrel{s_m}{\Longrightarrow}_{\mathrm{C}} \delta_{k_3^m}) = 25$ while $c(\bar{s}_{23} \xrightarrow{s_m} \delta_{h_0^m} \stackrel{\tau}{\Longrightarrow}_{\mathrm{C}} \delta_{h_2^m}) = 37$. It is routine to check the remaining pairs of states, thus $(\mathcal{A}_{32}, c) \lessapprox_c (\mathcal{A}_{23}, c)$.

Now, assume $(\mathcal{A}_{23}, c) \lessapprox_c (\mathcal{A}_{32}, c)$: by definition, it must hold that $\bar{s}_{32}\ \mathcal{C}\ \bar{s}_{23}$, so consider the transition $\bar{s}_{32} \xrightarrow{s_m} \delta_{k_0^m}$. For sure k_3^m and h_2^m are border states, as well as $\bar{s}_{32}$ and $\bar{s}_{23}$. Moreover, $\bar{s}_{32}$ and $\bar{s}_{23}$ can not be related by $\mathcal{W}$ to any other state as they are the only states performing s_m. Suppose that these are the only border states; this implies that $\bar{s}_{32} \xrightarrow{s_m} \delta_{k_0^m}$ has to be extended to $\bar{s}_{32} \xrightarrow{s_m} \delta_{k_0^m} \xRightarrow{\tau}_{\mathrm{C}} \delta_{k_3^m}$ whose cost is 25. The only possibility for $\bar{s}_{23}$ to match such transition while respecting the cost constraint is to perform the weak combined transition $\bar{s}_{23} \xRightarrow{s_m}_{\mathrm{C}} \delta_{h_i^m}$ with $i = 0$ or $i = 1$ and $k_3^m\ \mathcal{C}\ h_i^m$. Note that we can not use $\bar{s}_{23} \xRightarrow{s_m}_{\mathrm{C}} \delta_{h_2^m}$ since its cost is $37 \not\leq 25$. Independently on the chosen i, since $k_3^m\ \mathcal{C}\ h_i^m$ and $k_3^m \xrightarrow{r_m} \delta_{\bar{s}_{32}}$, h_i^m has to perform the weak combined transition $h_i^m \xRightarrow{r_m}_{\mathrm{C}} \delta_{\bar{s}_{23}}$ whose cost is $1 + 18 \cdot (2 - i) \not\leq 1$, so the condition is not satisfied. By applying the same approach to the case where we consider other states as border states, we can derive a similar failure, thus there does not exist any suitable cost relation $\mathcal{C}$ with $\bar{s}_{32}\ \mathcal{C}\ \bar{s}_{23}$, hence $(\mathcal{A}_{23}, c) \not\lessapprox_c (\mathcal{A}_{32}, c)$.

4.5 Decision Procedure

In order to algorithmically decide whether $(\mathcal{A}_1, c_1) \lessapprox_c (\mathcal{A}_2, c_2)$, we extend the polynomial decision procedure QUOTIENT that establishes whether $\mathcal{A}_1 \approx \mathcal{A}_2$ holds [11], to the MINORCOST algorithm depicted in Fig. 2 that computes $(\mathcal{W}, \mathcal{C})$ justifying $(\mathcal{A}_1, c_1) \lessapprox_c (\mathcal{A}_2, c_2)$: we first compute $\mathcal{W} = \text{QUOTIENT}(\mathcal{A}_1, \mathcal{A}_2)$ and then we consider as candidate cost relation $\mathcal{C} = \mathcal{C}'$ all pairs $s_2\ \mathcal{W}\ s_1$ with $s_2 \in S_2$ and $s_1 \in S_1$. In the main loop of MINORCOST we repeatedly refine $\mathcal{C}$ by removing all pairs that do not satisfy the conditions of Def. 9: if a check fails, we remove the offending pair (s_2, s_1) from $\mathcal{C}'$.

On termination of the loop, $\mathcal{C}$ contains only pairs satisfying Def. 9, so deciding whether $(\mathcal{A}_1, c_1) \lessapprox_c (\mathcal{A}_2, c_2)$ reduces to check whether $\bar{s}_2\ \mathcal{C}\ \bar{s}_1$ and whether for each $s_2 \in S_2$ there exists $s_1 \in S_1$ such that $s_2\ \mathcal{C}\ s_1$.

Given two *CPA*s $(\mathcal{A}_1, c_1)$ and $(\mathcal{A}_2, c_2)$, let $N = \max\{|S_1 \uplus S_2|, |D_1 \uplus D_2|\}$. Computing $\mathcal{W} = \text{QUOTIENT}(\mathcal{A}_1, \mathcal{A}_2)$ is polynomial in N (cf. [11, Thm. 11]), say $P(N)$; in the worst case, that occurs when we remove all pairs from $\mathcal{C}$, the main loop of MINORCOST is performed at most N^2 times; according to Thm. 1 and its corollaries, finding $\rho_2 \in \mathrm{Disc}(\mathcal{B}(\mathcal{W}) \cap S_2)$ such that $\mu_2 \xRightarrow{\tau}_{\mathrm{C}} \rho_2$ and $c_2(\mu_2 \xRightarrow{\tau}_{\mathrm{C}} \rho_2) = \min\{\, c_2(\mu_2 \xRightarrow{\tau}_{\mathrm{C}} \rho) \mid \rho \in \mathrm{Disc}(\mathcal{B}(\mathcal{W}) \cap S_2) \,\}$ is polynomial in N, say $R(N)$, by solving the LP problem $\min_c \mu_2 \xRightarrow{\tau}_{\mathrm{C}} \diamond\, \mathcal{L}(\mathcal{B})\ \delta_{b_2}$ where $b_2 \in \mathcal{B}(\mathcal{W}) \cap S_2$ and $\mathcal{B}$ is the reflexive, symmetric, and transitive closure of $\mathcal{B}(\mathcal{W})$. Similarly, $R(N)$ is also the complexity of either finding $\rho_1 \in \mathrm{Disc}(\mathcal{B}(\mathcal{W}) \cap S_1)$ such that $s_1 \xRightarrow{a}_{\mathrm{C}} \rho_1, \rho_2\ \mathcal{L}(\mathcal{C})\ \rho_1$, and $c_1(s_1 \xRightarrow{a}_{\mathrm{C}} \rho_1) \leq c_2(s_2 \xrightarrow{a} \mu_2 \xRightarrow{\tau}_{\mathrm{C}} \rho_2)$, or finding $\mu_1 \in \mathrm{Disc}(S_1)$ such that $s_1 \xRightarrow{a}_{\mathrm{C}} \mu_1, \mu_2\ \mathcal{L}(\mathcal{C})\ \mu_1$, and $c_1(s_1 \xRightarrow{a}_{\mathrm{C}} \mu_1) \leq c_2(s_2 \xrightarrow{a} \mu_2)$. This implies that the total complexity of MINORCOST is $P(N) + N^2 \cdot 2R(N)$.

Theorem 2. *Given two CPAs $(\mathcal{A}_1, c_1)$ and $(\mathcal{A}_2, c_2)$, checking $(\mathcal{A}_1, c_1) \lessapprox_c (\mathcal{A}_2, c_2)$ is polynomial in $N = \max\{|S_1 \uplus S_2|, |D_1 \uplus D_2|\}$.*

Regarding weak probabilistic cost-preserving bisimulation, the algorithm is actually simpler, since in order to check for the existence of weak combined transitions with a given cost $\mathbf{c}$, it is enough to add the new constraint $\sum_{(x,y)\in E} c_f((x,y)) \cdot f_{x,y} = \mathbf{c}$ to the $\min_c t \xRightarrow{a}_{\mathrm{C}} \diamond\, \mathcal{L}(\mathcal{R})\ \mu$ LP problem. This allows us to check in polynomial time whether

MINORCOST$((\mathcal{A}_1, c_1), (\mathcal{A}_2, c_2))$

$\mathcal{W} = \text{QUOTIENT}(\mathcal{A}_1, \mathcal{A}_2)$
$\mathcal{C}' = \mathcal{W} \cap (S_2 \times S_1)$
repeat
  $\mathcal{C} = \mathcal{C}'$
  for all $(s_2, a, \mu_2) \in D_2$ and $s_1 \in S_1$ s. t. $s_2 \; \mathcal{C}' \; s_1$ **do**
    if there exists $\bar{\rho}_2 \in \text{Disc}(\mathcal{B}(\mathcal{W}) \cap S_2)$ such that $\mu_2 \stackrel{\tau}{\Longrightarrow}_{\text{C}} \bar{\rho}_2$ **then**
      if Condition 1 of Def. 9 is not satisfied **then** $\mathcal{C}' = \mathcal{C}' \setminus \{(s_2, s_1)\}$
    else
      if Condition 2 of Def. 9 is not satisfied **then** $\mathcal{C}' = \mathcal{C}' \setminus \{(s_2, s_1)\}$
until $\mathcal{C}' \neq \mathcal{C}$
return $(\mathcal{W}, \mathcal{C})$

QUOTIENT$(\mathcal{A}_1, \mathcal{A}_2)$

$\mathcal{W} = \{S_1 \uplus S_2\}$;
$(\mathcal{C}, a, \mu) = \text{FINDSPLIT}(\mathcal{W})$;
while $\mathcal{C} \neq \emptyset$ **do**
  $\mathcal{W} = \text{REFINE}(\mathcal{W}, (\mathcal{C}, a, \mu))$;
  $(\mathcal{C}, a, \mu) = \text{FINDSPLIT}(\mathcal{W})$;
return $\mathcal{W}$

FINDSPLIT$(\mathcal{W})$

for all $(s, a, \mu) \in D = D_1 \uplus D_2$ **do**
  for all $t \in [s]_{\mathcal{W}}$ **do**
    if $t \stackrel{a}{\Longrightarrow}_{\text{C}} \diamond \mathcal{L}(\mathcal{W}) \; \mu$ has no solution **then**
      return $([s]_{\mathcal{W}}, a, \mu)$
return $(\emptyset, \tau, \delta_{\bar{s}})$

Fig. 2. Minor cost weak bisimulation decision procedure

two *CPA*s are weak probabilistic cost-preserving bisimilar: we compute QUOTIENT on the two *CPA*s where we have replaced in procedure FINDSPLIT the test for feasibility of $t \stackrel{a}{\Longrightarrow}_{\text{C}} \diamond \mathcal{L}(\mathcal{W}) \; \mu$ with the test for feasibility of $\min_c t \stackrel{a}{\Longrightarrow}_{\text{C}} \diamond \mathcal{L}(\mathcal{R}) \; \mu$ extended with the constraint $\sum_{(x,y) \in E} c_f((x, y)) \cdot f_{x,y} = c(s \stackrel{a}{\longrightarrow} \mu_s)$.

Theorem 3. *Given two CPAs* $(\mathcal{A}_1, c_1)$ *and* $(\mathcal{A}_2, c_2)$*, checking* $(\mathcal{A}_1, c_1) \approx_c (\mathcal{A}_2, c_2)$ *is polynomial in* $N = \max\{|S_1 \uplus S_2|, |D_1 \uplus D_2|\}$.

5 Discussion

This section puts our work in the context of related work and also discusses other options to follow.

Givan, Dean and Greig [9] have introduced the idea of strong bisimilarity for *MDP*s with state and transition costs, together with algorithms for minimisation to the quotient model. The minimisation with respect to weak probabilistic bisimulation on *PA* has lately been discussed [7], and it remains to be investigated how the minimisation can be applied for the minor cost approach meaningfully. For the cost-preserving bisimilarity, the adaptations are straightforward, so we can indeed minimise with respect to weak transition costs.

Since *CPA*s are basically *MDP*s with transition costs only, it is interesting to discuss how state costs can be handled. Indeed it is possible to turn state costs to transition costs by moving them on incoming or outgoing transitions. The concrete choice makes a difference, because the labels of incoming and outgoing transitions generally differ. If already transitions costs were present prior to the move, we end up with a second cost structure. Multiple cost structures can indeed also be integrated into our setting rather easily, one just needs to take the minor cost for all structures in the decision problem.

For *MDP*s, multiple reward structures have been investigated [8] in the context of model checking, and our approach naturally combines with that. Chatterjee, Majumdar, and Henzinger [5] investigated them in a setting with discounting. In fact, our polynomial time LP approach can be extended to compute the minimum cost of discounted weak combined transitions, if we can assume a polynomially bounded number of internal steps. Conversely, one can compute an upper bound on discounted but non-polynomially bounded weak combined transitions in polynomial time.

If discounting is integrated into the weak bisimulation definitions we propose, this however induces difficult-to-grasp equalities. This is because sequences of internal transitions of different length are abstracted away by weak bisimilarity, but they would imply different discounts. For similar reasons, our cost model does by itself not talk about traces. As long as internal transitions carry nonzero costs, the definition of the cost of a weak trace is not obvious. Even if two executions fragments have the same trace, i.e., the same sequence of visible actions, different execution fragments usually have different costs when they involve different internal transitions, in particular after the last external action of the trace. Moreover, even if the execution fragment does not involve internal transitions, it can have different costs as resulting by the resolution of probabilistic and nondeterministic choices, the latter performed by the scheduler.

Still, cost-preserving bisimilarity implies equal trace costs, and if $(\mathcal{A}_1, c_1)$ is in minor cost weak probabilistic bisimilarity with $(\mathcal{A}_2, c_2)$, then the trace costs of $(\mathcal{A}_1, c_1)$ are bounded from above by $(\mathcal{A}_2, c_2)$. Trace costs appear central in many cost related formalisms not involving probabilities, such as weighted timed and energy automata [3, 17], though without (internal) actions playing a dedicated role here, so it is worth to investigate trace costs in the *CPA* model as well.

While minor cost weak bisimilarity is implicitly asymmetric, we have still formulated it as an equivalence relation. The case study has demonstrated that this approach is undoubtedly useful. Yet, it seems worthwhile to also take inspiration from simulation and simulation distance approaches [1, 4] in this matter.

6 Concluding Remarks

In this paper we have presented the extension of Probabilistic Automata to Cost Probabilistic Automata and we have proposed two cost related weak probabilistic bisimulations: a cost preserving bisimulation and a cost bounding variation, minor cost weak probabilistic bisimulation, where the defender matches a transition with a cost that is bounded by at most the cost of the challenger.

Moreover we have shown how to compute in polynomial time the minimum cost for each transition, and hence to decide the two relations. Since the *CPA* model encompasses *MDP*s, the results apply readily to these models as well. In the future we plan to investigate how the compositionality properties of weak probabilistic bisimilarity extend from *PA* to *CPA*. With this, we aim to arrive at compositional construction and minimisation techniques that can be rolled out to operations research, automated planning, and decision support applications.

Acknowledgements. This work is supported by the DFG/NWO bilateral research programme ROCKS, by the DFG as part of the SFB/TR 14 AVACS, by the EU FP7 Programme under grant agreement no. 295261 (MEALS) and 318490 (SENSATION).

Andrea Turrini has received support from the Cluster of Excellence "Multimodal Computing and Interaction" (MMCI), part of the German Excellence Initiative.

References

1. Avni, G., Kupferman, O.: Making weighted containment feasible: A heuristic based on simulation and abstraction. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 84–99. Springer, Heidelberg (2012)
2. Baier, C., Engelen, B., Majster-Cederbaum, M.: Deciding bisimilarity and similarity for probabilistic processes. J. Computer and Systes Science 60(1), 187–231 (2000)
3. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Srba, J.: Infinite runs in weighted timed automata with energy constraints. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 33–47. Springer, Heidelberg (2008)
4. Černý, P., Henzinger, T.A., Radhakrishna, A.: Simulation distances. TCS 413(1), 21–35 (2012)
5. Chatterjee, K., Majumdar, R., Henzinger, T.A.: Markov decision processes with multiple objectives. In: Durand, B., Thomas, W. (eds.) STACS 2006. LNCS, vol. 3884, pp. 325–336. Springer, Heidelberg (2006)
6. Chehaibar, G., Garavel, H., Mounier, L., Tawbi, N., Zulian, F.: Specification and verification of the PowerScale $^{\mathrm{TM}}$ bus arbitration protocol: An industrial experiment with LOTOS. In: FORTE. pp. 435–450 (1996)
7. Eisentraut, C., Hermanns, H., Schuster, J., Turrini, A., Zhang, L.: The quest for minimal quotients for probabilistic automata. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 16–31. Springer, Heidelberg (2013)
8. Etessami, K., Kwiatkowska, M., Vardi, M.Y., Yannakakis, M.: Molti-objective model checking of Markov decision processes. Logical Methods in Computer Science 4(8), 1–21 (2008)
9. Givan, R., Dean, T., Greig, M.: Equivalence notions and model minimization in Markov decision processes. Artificial Intelligence 147(1-2), 163–223 (2003)
10. Hermanns, H., Katoen, J.P.: Automated compositional Markov chain generation for a plain-old telephone system. Science of Computer Programming 36(1), 97–127 (2000)
11. Hermanns, H., Turrini, A.: Deciding probabilistic automata weak bisimulation in polynomial time. In: FSTTCS, pp. 435–447 (2012)
12. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
13. Howard, R.A.: Dynamic Probabilistic Systems, Volume II: Semi-Markov and Decision Processes. Dover Publications (2007)
14. Jonsson, B., Larsen, K.G.: Specification and refinement of probabilistic processes. In: LICS, pp. 266–277 (1991)
15. Katoen, J.-P., Kemna, T., Zapreev, I., 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. Lynch, N.A., Segala, R., Vaandrager, F.W.: Observing branching structure through probabilistic contexts. SIAM J. on Computing 37(4), 977–1013 (2007)
17. Quaas, K.: Weighted timed MSO logics. In: Diekert, V., Nowotka, D. (eds.) DLT 2009. LNCS, vol. 5583, pp. 419–430. Springer, Heidelberg (2009)
18. Segala, R.: Modeling and Verification of Randomized Distributed Real-Time Systems. Ph.D. thesis. MIT (1995)
19. Segala, R.: Probability and nondeterminism in operational models of concurrency. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 64–78. Springer, Heidelberg (2006)

Compositional Verification and Optimization of Interactive Markov Chains*

Holger Hermanns[1], Jan Krčál[2], and Jan Křetínský[2,3]

[1] Saarland University – Computer Science, Saarbrücken, Germany
[2] Faculty of Informatics, Masaryk University, Czech Republic
[3] Institut für Informatik, Technical University Munich, Germany

Abstract. Interactive Markov chains (IMC) are compositional behavioural models extending labelled transition systems and continuous-time Markov chains. We provide a framework and algorithms for compositional verification and optimization of IMC with respect to time-bounded properties. Firstly, we give a specification formalism for IMC. Secondly, given a time-bounded property, an IMC component and the assumption that its unknown environment satisfies a given specification, we synthesize a scheduler for the component optimizing the probability that the property is satisfied in any such environment.

1 Introduction

The ever increasing complexity and size of systems together with software reuse strategies naturally enforce the need for component based system development. For the same reasons, checking reliability and optimizing performance of such systems needs to be done in a *compositional* way. The task is to get useful guarantees on the behaviour of a component of a larger system. The key idea is to incorporate assumptions on the rest of the system into the verification process. This *assume-guarantee reasoning* is arguably a successful divide-and-conquer technique in many contexts [MC81, AH96, HMP01].

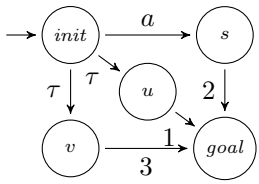
In this work, we consider a continuous-time stochastic model called *interactive Markov chains* (IMC). First, we give a language for expressing assumptions about IMC. Second, given an IMC, an assumption on its environment and a property of interest, we synthesize a controller of the IMC that optimizes the guarantee, and we compute this optimal guarantee, too.

Interactive Markov chains are behavioural models of probabilistic systems running in continuous real time appropriate for the component-based approach [HK09]. IMC have a well-understood compositional theory rooted in process algebra, and are in use as semantic backbones for dynamic fault trees, architectural

* The work has received support from the Czech Science Foundation, project No. P202/12/G061, from the German Science Foundation DFG as part of SFB/TR 14 AVACS, and by the EU FP7 Programme under grant agreement no. 295261 (MEALS) and 318490 (SENSATION).

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 364–379, 2013.

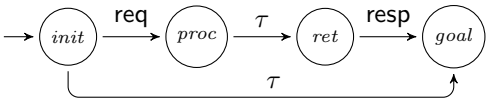
description languages, generalized stochastic Petri nets and Statemate extensions, see [HK09] for a survey. IMC are applied in a large spectrum of practical applications, ranging from water treatment facilities [HKR+10] to ultra-modern satellite designs [EKN+12].



IMC arise from classical labelled transition systems by incorporating the possibility to change state according to a random *delay* governed by a negative exponential distribution with a given rate, see transitions labelled 1, 2 and 3 in the figure. Apart from delay expirations, state transitions may be triggered by the execution of *internal* (τ) actions or *external* (synchronization) actions. Internal actions are assumed to happen instantaneously and therefore take precedence over delay transitions. External actions are the process algebraic means for interaction with other components, see a in the figure. By dropping the delay transitions, labelled transition systems are regained in their entirety. Dropping action-labelled transitions instead yields continuous-time Markov chains – one of the most used performance and reliability models.

The fundamental problem in the analysis of IMC is that of *time-bounded reachability*. It is the problem to approximate the probability that a given set of states is reached within a given deadline. We illustrate the compositional setting of this problem in the following examples.

Examples. In the first example, consider the IMC $\mathcal{C}$ from above and an unknown environment $\mathcal{E}$ with no assumptions. Either $\mathcal{E}$ is initially not ready to synchronize on the external action a and thus one of the internal actions is taken, or $\mathcal{E}$ is willing to synchronize on a at the beginning. In the latter case, whether τ or a happens is resolved non-deterministically. Since this is out of control of $\mathcal{C}$, we must assume the worst case and let the environment decide which of the two options will happen. For more details on this design choice, see [BHK+12]. If there is synchronization on a, the probability to reach *goal* within time $t = 1.5$ is $1 - e^{-2t} \approx 0.95$. Otherwise, $\mathcal{C}$ is given the choice to move to u or v. Naturally, v is the choice maximizing the chance to get to *goal* on time as it has a higher rate associated. In this case the probability amounts to $1 - e^{-3t} \approx 0.99$, while if u were chosen, it would be only 0.78. Altogether, the guaranteed probability is 95% and the strategy of $\mathcal{C}$ is to choose v in *init*.



The example depicted on the right illustrates the necessity of assumptions on the environment: As it is, the environment can drive the component to state *ret* and let it get stuck there by not synchronising on resp ever. Hence no better guarantee than 0 can be derived. However, this changes if we know some specifics about the behaviour of the environment: Let us assume that we know that once synchronization on req occurs, the environment must be ready to synchronise on resp within some random time according to, say, an exponential distribution with rate 2. Under this assumption, we are able to derive a guarantee of 95%, just as in the previous example.

Observe the form of the time constraint we imposed in the last example: "within a random time distributed according to Exp(2)" or symbolically $\Diamond_{\leq Exp(2)} \varphi$. We call this a *continuous time constraint*. If a part of the environment is e.g. a model of a communication network, it is clear we cannot impose hard bounds (discrete time constraints) such as "within 1.5" as in e.g. a formula of MTL $\Diamond_{\leq 1.5}\varphi$. Folklore tells us that messages might get delayed for longer than that. Yet we want to express high assurance that they arrive on time. In this case one might use e.g. a formula of CSL $Pr_{\geq 0.95}(\Diamond_{\leq 1.5}\varphi)$. However, consider now a system with two transitions labelled with resp in a row. Then this CSL formula yields only a zero guarantee. By splitting the time 1.5 in halves, the respective $Pr_{\geq 0.77}(\Diamond_{\leq 0.75}\varphi)$ yields only the guarantee $0.77^2 = 0.60$. The actual guarantee 0.80 is given by the convolution of the two exponential distributions and as such can be exactly obtained from our continuous time constraint $\Diamond_{\leq Exp(2)}\varphi$.

Our contribution is the following:

1. We introduce a specification formalism to express assumptions on continuous-time stochastic systems. The novel feature of the formalism are the continuous time constraints, which are vital for getting guarantees with respect to time-bounded reachability in IMC.
2. We incorporate the assume-guarantee reasoning to the IMC framework. We show how to synthesize ϵ-optimal schedulers for IMC in an *unknown environment satisfying a given specification* and approximate the respective guarantee.

In our recent work [BHK$^+$12] we considered a very restricted setting of the second point. Firstly, we considered no assumptions on the environment as the environment of a component might be entirely unknown in many scenarios. Secondly, we were restricted to IMC that never enable internal and external transitions at the same state. This was also a severe limitation as this property is not preserved during the IMC composition process and restricts the expressivity significantly. Both examples above violate this assumption. In this paper, we lift the assumption.

Each of the two extensions shifts the solution methods from complete information stochastic games to (one-sided) *partial observation* stochastic games, where we need to solve the quantitative reachability problem. While this is undecidable in general, we reduce our problem to a game played on an *acyclic graph* and show how to solve our problem in exponential time. (Note that even the qualitative reachability in the acyclic case is PSPACE-hard [CD10].)

Related Work. The *synthesis* problem is often stated as a game where the first player controls a component and the second player simulates an environment [RW89]. Model checking of *open* systems, i.e. operating in an unknown environment, has been proposed in [KV96]. There is a body of work on *assume-guarantee* reasoning for parallel composition of *real-time* systems [TAKB96, HMP01]. Lately, games with *stochastic continuous-time* have gained attention, for a very general class see [BF09]. While the second player models possible schedulers of the environment, the structure of the environment

is fixed there and the verification is thus not compositional. The same holds for [Spr11, HNP+11], where time is under the control of the components.

A compositional framework requires means for specification of systems. A specification can be also viewed as an *abstraction* of a set of systems. Three valued abstractions stemming from [LT88] have also been applied to the timed setting, namely in [KKLW07] to continuous-time Markov chains (IMC with no non-determinism), or in [KKN09] to IMC. Nevertheless, these abstractions do not allow for constraints on time distributions. Instead they would employ abstractions on transition probabilities. Further, a compositional framework with timed specifications is presented in [DLL+12]. This framework explicitly allows for time constraints. However, since the systems under consideration have non-deterministic flow of time (not stochastic), the natural choice was to only allow for discrete (not continuous) time constraints.

Although IMC support compositional design very well, analysis techniques for IMC proposed so far (e.g. [KZH+11, KKN09, ZN10, GHKN12] are not compositional. They are all bound to the assumption that the analysed IMC is a *closed* system, i.e. it does not depend on interaction with the environment (all actions are internal). Some preliminary steps to develop a framework for synthesis of controllers based on models of hardware and control requirements have been taken in [Mar11]. The first attempt at compositionality is our very recent work [BHK+12] discussed above.

Algorithms for the *time-bounded reachability* problem for closed IMC have been given in [ZN10, BS11, HH13] and compositional abstraction techniques to compute it are developed in [KKN09]. In the closed interpretation, IMC have some similarities with continuous-time Markov decision processes. For this formalism, algorithms for time-bounded reachability are developed in [BHKH05, BS11].

2 Interactive Markov Chains

In this section, we introduce the formalism of interactive Markov chains together with the standard way to compose them. We denote by $\mathbb{N}$, $\mathbb{R}_{>0}$, and $\mathbb{R}_{\geq 0}$ the sets of positive integers, positive real numbers and non-negative real numbers, respectively. Further, let $\mathcal{D}(S)$ denote the set of probability distributions over the set S.

Definition 1 (IMC). *An* interactive Markov chain (IMC) *is a quintuple* $\mathcal{C} = (S, \mathbb{A}\mathrm{ct}^\tau, \hookrightarrow, \rightsquigarrow, s_0)$ *where* S *is a finite set of* states, $\mathbb{A}\mathrm{ct}^\tau$ *is a finite set of* actions *containing a designated* internal action τ, $s_0 \in S$ *is an* initial *state,*

- $\hookrightarrow \subseteq S \times \mathbb{A}\mathrm{ct}^\tau \times S$ *is an* interactive transition *relation, and*
- $\rightsquigarrow \subseteq S \times \mathbb{R}_{>0} \times S$ *is a* Markovian transition *relation.*

Elements of $\mathbb{A}\mathrm{ct} := \mathbb{A}\mathrm{ct}^\tau \setminus \{\tau\}$ are called *external actions*. We write $s \overset{a}{\hookrightarrow} t$ whenever $(s, a, t) \in \hookrightarrow$, and $s \overset{\lambda}{\rightsquigarrow} t$ whenever $(s, \lambda, t) \in \rightsquigarrow$ where λ is called a *rate* of the transition. We say that an external action a, or internal τ, or Markovian transition is *available* in s, if $s \overset{a}{\hookrightarrow} t$, $s \overset{\tau}{\hookrightarrow} t$ or $s \overset{\lambda}{\rightsquigarrow} t$ for some t (and λ), respectively.

IMC are well suited for compositional modelling, where systems are built out of smaller ones using standard composition operators. *Parallel composition* $\|_A$ over a *synchronization alphabet* A produces a product of two IMC with transitions given by the rules

(PC1) $(s_1, s_2) \overset{a}{\hookrightarrow} (s_1', s_2')$ for each $s_1 \overset{a}{\hookrightarrow} s_1'$ and $s_2 \overset{a}{\hookrightarrow} s_2'$ and $a \in A$,
(PC2, PC3) $(s_1, s_2) \overset{a}{\hookrightarrow} (s_1', s_2)$ for each $s_1 \overset{a}{\hookrightarrow} s_1'$ and $a \notin A$, and symmetrically,
(PC4, PC5) $(s_1, s_2) \overset{\lambda}{\rightsquigarrow} (s_1', s_2)$ for each $s_1 \overset{\lambda}{\rightsquigarrow} s_1'$, and symmetrically.

Further, *hiding* $\setminus A$ an alphabet A, yields a system, where each $s \overset{a}{\hookrightarrow} s'$ with $a \notin A$ is left as it is, and each $s \overset{a}{\hookrightarrow} s'$ with $a \in A$ is replaced by internal $s \overset{\tau}{\hookrightarrow} s'$.

Hiding $\setminus \mathbb{A}\mathrm{ct}$ thus yields a *closed* IMC, where external actions do not appear as transition labels (i.e. $\hookrightarrow \subseteq S \times \{\tau\} \times S$). A closed IMC (under a scheduler σ, see below) moves from state to state and thus produces a *run* which is an infinite sequence of the form $s_0\, t_1\, s_1\, t_2\, s_2 \cdots$ where s_n is the n-th visited state and t_n is the time of arrival to s_n. After n steps, the scheduler resolves the non-determinism among internal τ transitions based on the *path* $\mathfrak{p} = s_0\, t_1 \cdots t_n\, s_n$.

Definition 2 (Scheduler). *A* scheduler *of an IMC* $\mathcal{C} = (S, \mathbb{A}\mathrm{ct}^\tau, \hookrightarrow, \rightsquigarrow, s_0)$ *is a measurable function* $\sigma : (S \times \mathbb{R}_{\geq 0})^* \times S \to \mathcal{D}(S)$ *such that for each path* $\mathfrak{p} = s_0\, t_1\, s_1 \cdots t_n\, s_n$ *with* s_n *having* τ *available,* $\sigma(\mathfrak{p})(s) > 0$ *implies* $s_n \overset{\tau}{\hookrightarrow} s$. *The set of all schedulers for* $\mathcal{C}$ *is denoted by* $\mathfrak{S}(\mathcal{C})$.

The decision of the scheduler $\sigma(\mathfrak{p})$ determines t_{n+1} and s_{n+1} as follows. If s_n has available τ, then the run proceeds immediately, i.e. at time $t_{n+1} := t_n$, to a state s_{n+1} randomly chosen according to the distribution $\sigma(\mathfrak{p})$. Otherwise, only Markovian transitions are available in s_n. In such a case, after waiting for a random time t chosen according to the exponential distribution with the rate $\mathrm{R}(s_n) = \sum_{s_n \overset{\lambda}{\rightsquigarrow} s'} \lambda$, the run moves at time $t_{n+1} := t_n + t$ to a randomly chosen next state s_{n+1} with probability λ / r where $s_n \overset{\lambda}{\rightsquigarrow} s_{n+1}$. This defines a probability space $(\mathbb{R}\mathrm{uns}, \mathcal{F}, \mathcal{P}_{\mathcal{C}}^{\sigma})$ over the runs in the standard way [ZN10].

3 Time-Bounded Reachability

In this section, we introduce the studied problems. One of the fundamental problems in verification and performance analysis of continuous-time stochastic systems is time-bounded reachability. Given a *closed* IMC $\mathcal{C}$, a set of goal states $G \subseteq S$ and a time bound $T \in \mathbb{R}_{\geq 0}$, the *value of time-bounded reachability* is defined as $\sup_{\sigma \in \mathfrak{S}(\mathcal{C})} \mathcal{P}_{\mathcal{C}}^{\sigma}\left[\Diamond^{\leq T} G\right]$ where $\mathcal{P}_{\mathcal{C}}^{\sigma}\left[\Diamond^{\leq T} G\right]$ denotes the probability that a run of $\mathcal{C}$ under the scheduler σ visits a state of G before time T. We have seen an example in the introduction. A standard assumption over all analysis techniques published for IMC [KZH+11, KKN09, ZN10, GHKN12] is that each cycle contains a Markovian transition. It implies that the probability of taking infinitely many transitions in finite time, i.e. of Zeno behaviour, is zero. One can ε-approximate the value and compute the respective scheduler in time $\mathcal{O}(\lambda^2 T^2 / \varepsilon)$ [ZN10] recently improved to $\mathcal{O}(\sqrt{\lambda^3 T^3 / \varepsilon})$ [HH13].

For an *open* IMC to be put in parallel with an unknown environment, the optimal scheduler is computed so that it optimizes the guarantee against all possible environments. Formally, for an IMC $\mathcal{C} = (C, \mathbb{A}\mathrm{ct}^\tau, \hookrightarrow, \rightsquigarrow, c_0)$ and an environment IMC $\mathcal{E}$ with the same action alphabet $\mathbb{A}\mathrm{ct}^\tau$, we introduce a composition $\mathcal{C}|\mathcal{E} = (\mathcal{C} \parallel_{\mathbb{A}\mathrm{ct}} \mathcal{E}) \setminus \mathbb{A}\mathrm{ct}$ where all open actions are hidden, yielding a closed system. In order to compute guarantees on $\mathcal{C}|\mathcal{E}$ provided we use a scheduler σ in $\mathcal{C}$, we consider schedulers π of $\mathcal{C}|\mathcal{E}$ that *respect* σ on the internal actions of $\mathcal{C}$, written $\pi \in \mathfrak{S}_\sigma(\mathcal{C}|\mathcal{E})$; the formal definition is below. The *value of compositional time-bounded reachability* is then defined in [BHK+12] as

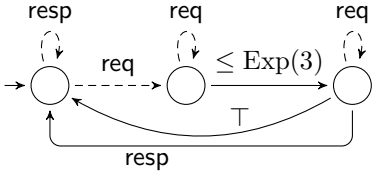
$$\sup_{\sigma \in \mathfrak{S}(\mathcal{C})} \inf_{\substack{\mathcal{E} \in \mathrm{ENV} \\ \pi \in \mathfrak{S}_\sigma(\mathcal{C}|\mathcal{E})}} \mathcal{P}^\pi_{\mathcal{C}|\mathcal{E}}\left[\Diamond^{\leq T} G\right]$$

where ENV denotes the set of all IMC with the action alphabet $\mathbb{A}\mathrm{ct}^\tau$ and $\Diamond^{\leq T} G$ is the set of runs that reach G in the first component before T. Now π *respects* σ on internal actions of $\mathcal{C}$ if for every path $\mathfrak{p} = (c_0, e_0)\, t_1 \cdots t_n (c_n, e_n)$ of $\mathcal{C}|\mathcal{E}$ there is $p \in [0,1]$ such that for each internal transition $c_n \overset{\tau}{\hookrightarrow} c$ of $\mathcal{C}$, we have $\pi(\mathfrak{p})(c, e_n) = p \cdot \sigma(\mathfrak{p}_\mathcal{C})(c)$. Here $\mathfrak{p}_\mathcal{C}$ is the projection of $\mathfrak{p}$ where σ can only see the path of moves in $\mathcal{C}$ and not in which states $\mathcal{E}$ is. Formally, we define *observation* of a path $\mathfrak{p} = (c_0, e_0)\, t_1 \cdots t_n (c_n, e_n)$ as $\mathfrak{p}_\mathcal{C} = c_0 t_1 \cdots t_n c_n$ where each maximal consecutive sequence $t_i\, c_i \cdots t_j\, c_j$ with $c_k = c_i$ for all $i \leq k \leq j$ is rewritten to $t_i\, c_i$. This way, σ ignores precisely the internal steps of $\mathcal{E}$.

3.1 Specifications of Environments

In the second example in the introduction, without any assumptions on the environment only zero guarantees could be derived. The component was thus indistinguishable from an entirely useless one. In order to get a better guarantee, we introduce a formalism to specify assumptions on the behaviour of environments.

Example 1. In the mentioned example, if we knew that after an occurrence of req the environment is ready to synchronize on resp in time distributed according to Exp(3) or faster, we would be able to derive a guarantee of 0.26. We will depict this assumption as shown below.



The dashed arrows denote *may* transitions, which may or may not be available, whereas the full arrows denote *must* transitions, which the environment is ready to synchronize on. Full arrows are further used for time transitions.

Although such a system resembles a timed automaton, there are several fundamental differences. Firstly, the time constraints are given by probability distributions instead of constants. Secondly, there is only one clock that, moreover, gets reset whenever the state is *changed*. Thirdly, we allow modalities of may and must transitions. Further, as usual with timed or stochastic specifications, we require determinism.

Definition 3 (MCA syntax). *A* continuous time constraint *is either* $\top$ *or of the form* $\bowtie d$ *with* $\bowtie \in \{\leq, \geq\}$ *and* d *a continuous distribution. We denote the set of all continuous time constraints by* $\mathcal{CTC}$*. A* modal continuous-time automaton (MCA) *over* Σ *is a tuple* $\mathcal{S} = (Q, q_0, \dashrightarrow, \longrightarrow, \rightsquigarrow)$*, where*

- Q *is a non-empty finite set of* locations *and* $q_0 \in Q$ *is an* initial location*,*
- $\longrightarrow, \dashrightarrow : Q \times \Sigma \to Q$ *are* must *and* may *transition functions, respectively, satisfying* $\longrightarrow \subseteq \dashrightarrow$*,*
- $\rightsquigarrow : Q \to \mathcal{CTC} \times Q$ *is a* time flow *function.*

We have seen an example of an MCA in the previous example. Note that upon taking req from the first state, the waiting time is chosen and the waiting starts. On the other hand, when req *self-loop* is taken in the middle state, the waiting process is not restarted, but continues on the background independently.(1) We introduce this independence as a useful feature to model properties as "response follows within some time after request" in the setting with concurrently running processes. Further, we have transitions under $\top$ corresponding to "> 0", meaning there is no restriction on the time distribution except that the transition takes non-zero time. We formalize this in the following definition. With other respects, the semantics of may and must transitions follows the standards of modal transition systems [LT88].

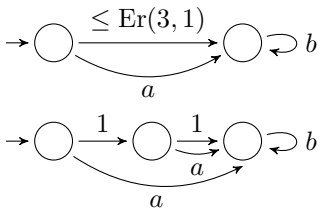
Definition 4 (MCA semantics). *An IMC* $\mathcal{E} = (E, \mathbb{A}\mathrm{ct}^\tau, \hookrightarrow, \rightsquigarrow, e_0)$ conforms *to an MCA specification* $\mathcal{S} = (Q, q_0, \dashrightarrow, \longrightarrow, \rightsquigarrow)$*, written* $\mathcal{E} \models \mathcal{S}$*, if there is a* satisfaction relation $\mathcal{R} \subseteq E \times Q$ *containing* (e_0, q_0) *and satisfying for each* $(e, q) \in \mathcal{R}$ *that whenever*

1. $q \xrightarrow{a} q'$ *then there is some* $e \overset{a}{\hookrightarrow} e'$ *and if, moreover,* $q \neq q'$ *then* $e' \mathcal{R} q'$*,*
2. $e \overset{a}{\hookrightarrow} e'$ *then there is (unique)* $q \overset{a}{\dashrightarrow} q'$ *and if, moreover,* $q \neq q'$ *then* $e' \mathcal{R} q'$*,*
3. $e \overset{\tau}{\hookrightarrow} e'$ *then* $e' \mathcal{R} q$*,*
4. $q \overset{ctc}{\rightsquigarrow} q'$ *then for every IMC* $\mathcal{C}$ *and every scheduler* $\pi \in \mathfrak{S}(\mathcal{C}|e)$,(2) *there is a random variable* $Stop : \mathbb{R}\mathrm{uns} \to \mathbb{R}_{>0}$ *on the probability space* $(\mathbb{R}\mathrm{uns}, \mathcal{F}, \mathcal{P}^\pi_{\mathcal{C}|e})$ *such that*
   - *if ctc is of the form* $\bowtie d$ *then the cumulative distribution function of Stop is point-wise* $\bowtie$ *cumulative distribution function of* d *(there are no constraints when* $ctc = \top$*), and*
   - *for every run* ρ *of* $\mathcal{C}|e$ *under* π*, either a transition corresponding to synchronization on action* a *with* $q \overset{a}{\dashrightarrow} q' \neq q$ *is taken before time* $Stop(\rho)$*, or*
     - *the state* (c, e') *visited at time* $Stop(\rho)$ *satisfies* $e' \mathcal{R} q'$*, and*
     - *for all states* $(\bar{c}, \bar{e})$ *visited prior to that, whenever*
       (a) $q \xrightarrow{a} q'$ *then there is* $e \overset{a}{\hookrightarrow} e'$*,*
       (b) $e \overset{a}{\hookrightarrow} e'$ *then there is* $q \overset{a}{\dashrightarrow} q'$*.*

The semantics of $\mathcal{S}$ *is the set* $[\![\mathcal{S}]\!] = \{\mathcal{E} \in \mathrm{IMC} \mid \mathcal{E} \models \mathcal{S}\}$ *of all conforming IMC.*

(1) This makes no difference for memoryless exponential distributions, but for all other distributions it does.

(2) Here e stands for the IMC $\mathcal{E}$ with the initial state e.



Example 2. We illustrate this definition. Consider the MCA on the right above specifying that a is ready and b will be ready either immediately after taking a or within the time distributed according to the Erlang distribution Er$(3, 1)$, which is a convolution of three Exp(1) distributions. The IMC below conforms to this specification (here, $Stop \sim Er(2, 1)$ can be chosen). However, observe that it would not conform, if there was no transition under a from the middle to the right state. Satisfying the modalities throughout the waiting is namely required by the last bullet of the previous definition.

3.2 Assume-Guarantee Optimization

We can now formally state what guarantees on time-bounded reachability we can derive provided the unknown environment conforms to a specification $\mathcal{S}$. Given an *open* IMC $\mathcal{C}$, a set of goal states $G \subseteq C$ and a time bound $T \in \mathbb{R}_{\geq 0}$, the *value of compositional time-bounded reachability conditioned by an MCA* $\mathcal{S}$ is defined as

$$v_{\mathcal{S}}(\mathcal{C}) \quad := \quad \sup_{\sigma \in \mathfrak{S}(\mathcal{C})} \inf_{\substack{\mathcal{E} \in \mathrm{ENV}: \mathcal{E} \models \mathcal{S} \\ \pi \in \mathfrak{S}_\sigma(\mathcal{C}|\mathcal{E})}} \mathcal{P}^{\pi}_{\mathcal{C}|\mathcal{E}}\left[\Diamond^{\leq T} G\right]$$

In this paper, we pose a technical assumption on the set of schedulers of $\mathcal{C}$. For some clock resolution $\delta > 0$, we consider only such schedulers σ that take the same decision for any pair of paths $c_0 t_1 \dots t_n c_n$ and $c_0 t'_1 \dots t'_n c_n$ with t_i and t'_i equal when rounded down to a multiple of δ for all $1 \leq i \leq n$. This is no practical restriction as it is not possible to achieve arbitrary resolution of clocks when implementing the scheduler. Observe this is a safe assumption as it is *not* imposed on the unknown environment.

We consider specifications $\mathcal{S}$ where distributions have differentiable density functions. In the rest of the paper we show how to approximate $v_{\mathcal{S}}(\mathcal{C})$ for such $\mathcal{S}$. Firstly, we make a product of the given IMC and MCA. Secondly, we transform the product to a game. This game is further discretized into a partially observable stochastic game played on a dag where the quantitative reachability is solved. For full proofs, see [HKK13].

4 Product of IMC and Specification

In this section, we first translate MCA $\mathcal{S}$ into a sequence of IMC $(\mathcal{S}_i)_{i \in \mathbb{N}}$. Second, we combine the given IMC $\mathcal{C}$ with the sequence $(\mathcal{S}_i)_{i \in \mathbb{N}}$ into a sequence of product IMC $(\mathcal{C} \times \mathcal{S}_i)_{i \in \mathbb{N}}$ that will be further analysed. The goal is to reduce the case where the unknown environment is bound by the specification to a setting where we solve the problem for the product IMC while quantifying over all possible environments (satisfying only a simple technical assumption discussed at the end of the section), denoted ENV$'$. The reason why we need a sequence of products

instead of one product is that we need to approximate arbitrary distributions with more and more precise and detailed hyper-Erlang distributions expressible in IMC. Formally, we want to define the sequence of the products $\mathcal{C} \times \mathcal{S}_i$ so that

$$v_{product}(\mathcal{C} \times \mathcal{S}_i) \quad := \quad \sup_{\sigma \in \mathfrak{S}(\mathcal{C})} \inf_{\substack{\mathcal{E} \in \mathrm{ENV}' \\ \pi \in \mathfrak{S}_\sigma((\mathcal{C} \times \mathcal{S}_i)|\mathcal{E})}} \mathcal{P}^\pi_{(\mathcal{C} \times \mathcal{S}_i)|\mathcal{E}} \left[\Diamond^{\leq T} G \right]$$

approximates the compositional value:

Theorem 1. *For every IMC $\mathcal{C}$ and MCA $\mathcal{S}$, $v_{\mathcal{S}}(\mathcal{C}) = \lim_{i \to \infty} v_{product}(\mathcal{C} \times \mathcal{S}_i)$.*

Note that in $v_{product}$, σ is a scheduler over $\mathcal{C}$, not the whole product $\mathcal{C} \times \mathcal{S}_i$.[3] Constructing a product with the specification intuitively corresponds to adding a known, but uncontrollable and unobservable part of the environment to $\mathcal{C}$. We proceed as follows: We translate the MCA $\mathcal{S}$ into a sequence of IMC $\mathcal{S}_i$ and then the product will be defined as basically a parallel composition of $\mathcal{C}$ and $\mathcal{S}_i$.

There are two steps in the translation of $\mathcal{S}$ to $\mathcal{S}_i$. Firstly, we deal with the modal transitions. A *may* transition under a is translated to a standard external transition under a that has to synchronize with a in both $\mathcal{C}$ and $\mathcal{E}$ simultaneously, so that the environment may or may not let the synchronization occur. Further, each *must* transition under a is replaced by an external transition, that synchronizes with a in $\mathcal{C}$, but is hidden before making product with the environment. This way, we guarantee that $\mathcal{C}$ can take a and make progress no matter if the general environment $\mathcal{E}$ would like to synchronize on a or not.

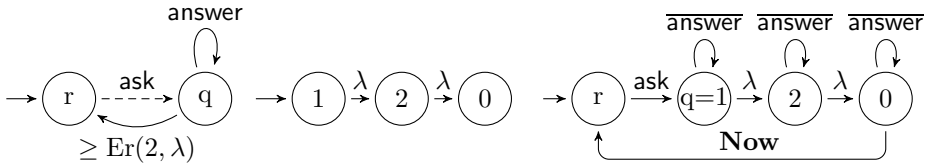
Formally, the must transitions are transformed into special "barred" transitions that will be immediately hidden in the product $\mathcal{C} \times \mathcal{S}_i$ as opposed to transitions arising from may transitions. Let $\overline{\mathbb{A}\mathrm{ct}} = \{\bar{a} \mid a \in \mathbb{A}\mathrm{ct}\}$ denote a fresh copy of the original alphabet. We replace all modal transitions as follows

- whenever $q \overset{a}{\dashrightarrow} r$ set $q \overset{a}{\hookrightarrow} r$,
- whenever $q \overset{a}{\longrightarrow} r$ set $q \overset{\bar{a}}{\hookrightarrow} r$.

The second step is to deal with the *timed* transitions, especially with the constraints of the form $\bowtie d$. Such a transition is, roughly speaking, replaced by a phase-type approximation of d. This is a continuous-time Markov chain (an IMC with only timed transitions) with a sink state such that the time to reach the sink state is distributed with d'. For any continuous distribution d, we can find such d' arbitrarily close to d.

Example 3. Consider the following MCA on the left. It specifies that whenever ask is taken, it cannot be taken again for at least the time distributed by $\mathrm{Er}(2, \lambda)$ and during all that time, it is ready to synchronize on answer. This specifies systems that are allowed to ask, but not too often, and whenever they ask, they must be ready to receive (possibly more) answers for at least the specified time.

[3] Here we overload the notation $\mathfrak{S}_\sigma((\mathcal{C} \times \mathcal{S}_i)|\mathcal{E})$ introduced for pairs in a straightforward way to triples, where σ ignores both the second and the third components.



After performing the first step of replacing the modal transitions as described above, we proceed with the second step as follows. We replace the timed transition with a phase-type, e.g. the one represented by the IMC in the middle. Observe that while the Markovian transitions are taken, answer must still be available. Hence, we duplicate the corresponding self-loops on all the new states. Further, since the time constraint is of the form $\geq$, getting to the state $(q, 0)$ does not guarantee that we already get to the state r. It can possibly take longer. To this end, we connect the states $(q, 0)$ and r by a special external action **Now**. Since this action is synchronized with $\mathcal{E} \in \mathrm{ENV}'$, the environment can block the progress for arbitrarily long time. Altogether, we obtain the IMC on the right.

In the case of "$\leq$" condition, we would instead add the **Now** transition from each auxiliary state to the sink, which could instead shorten the waiting time.

When constructing $\mathcal{S}_i$, we replace each distribution d with its hyper-Erlang phase-type approximation d_i with i branches of lengths 1 to i and rates $\sqrt{i}$ in each branch. For formal description, see [HKK13]. Formally, let $\mathbf{Now} \notin \mathbb{A}\mathrm{ct} \cup \overline{\mathbb{A}\mathrm{ct}}$ be a fresh action. We replace all timed transitions as follows:

- whenever $q \overset{\top}{\rightsquigarrow} r$ such that $q \neq r$ set $q \overset{\mathbf{Now}}{\hookrightarrow} r$,
- whenever $q \overset{\bowtie d}{\rightsquigarrow} r$ where the phase-type d_i corresponds to a continuous-time Markov chain (IMC with only timed transitions) with the set of states D, the initial state *1* and the sink state *0*, then
  1. identify the states q and *1*,
  2. for every $u \in D$ and $q \overset{\alpha}{\hookrightarrow} q$, set $u \overset{\alpha}{\hookrightarrow} u$,
  3. for every $u \in D$ and $q \overset{\alpha}{\hookrightarrow} p$ with $p \neq q$, set $u \overset{\alpha}{\hookrightarrow} p$,
  4. if $\bowtie = \leq$, then identify r and *0*, and set $u \overset{\mathbf{Now}}{\hookrightarrow} r$ for each $u \in D$,
  5. if $\bowtie = \geq$, then set $0 \overset{\mathbf{Now}}{\hookrightarrow} r$.

Intuitively, the new timed transitions model the delays, while in the "$\leq$" case, the action **Now** can be taken to speed up the process of waiting, and in the "$\geq$" case, **Now** can be used to block further progress even after the delay has elapsed.

The product is now the parallel composition of $\mathcal{C}$ and $\mathcal{S}_i$, where each action $\bar{a}$ synchronizes with a and the result is immediately hidden. Formally, the product $\mathcal{C} \times \mathcal{S}$ is defined as $\mathcal{C} \parallel^{\mathbf{PC6}}_{\mathbb{A}\mathrm{ct} \cup \overline{\mathbb{A}\mathrm{ct}}} \mathcal{S}_i$, where $\parallel^{\mathbf{PC6}}_{\mathbb{A}\mathrm{ct} \cup \overline{\mathbb{A}\mathrm{ct}}}$ is the parallel composition with one additional axiom:

(PC6) $s_1 \overset{a}{\hookrightarrow} s_1'$ and $s_2 \overset{\bar{a}}{\hookrightarrow} s_2'$ implies $(s_1, s_2) \overset{\tau}{\hookrightarrow} (s_1', s_2')$,

saying that a synchronizes also with $\bar{a}$ and, in that case, is immediately hidden (and any unused $\bar{a}$ transitions are thrown away).

The idea of **Now** is that it can be taken in arbitrarily short, but non-zero time. To this end, we define ENV′ in the definition of $v_{product}(\mathcal{C} \times \mathcal{S}_i)$ to denote

all environments where **Now** is only available in states that can be entered by only a Markovian transition. Due to this requirement, each **Now** can only be taken after waiting for some time.

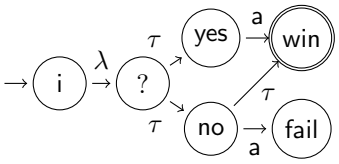
5 Controller-Environment Games

So far, we have reduced our problem to computing $\lim_{i\to\infty} v_{product}(\mathcal{C} \times \mathcal{S}_i)$. Note that we are still quantifying over unknown environments. Further, the behaviour of each environment is limited by the uncontrollable *stochastic* flow of time caused by its Markovian transitions. This setting is still too difficult to be solved directly. Therefore, in this section, we reduce this setting to one, where the stochastic flow of time of the environment (limited in an unknown way) is replaced by a free *non-deterministic* choice of the *second player*.

We want to turn the product IMC $\mathcal{C} \times \mathcal{S}_i$ into a two-player *controller–environment game* (CE game) $\mathcal{G}_i$, where player **con** controls the decisions over internal transitions in $\mathcal{C}$; and player **env** simulates the environment including speeding-up/slowing-down $\mathcal{S}$ using **Now** transitions. In essence, **con** chooses in each state with internal transitions one of them, and **env** chooses in each state with external (and hence synchronizing) transitions either which of them should be taken, or a *delay* $d \in \mathbb{R}_{>0}$ during which no synchronization occurs. The internal and external transitions take zero time to be executed if chosen. Otherwise, the game waits until either the delay d elapses or a Markovian transition occurs.

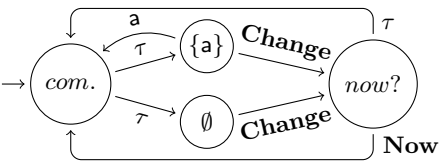
This is the approach taken in [BHK+12] where no specification is considered. However, there is a catch. This construction is only correct under the assumption of [BHK+12] that there are no states of $\mathcal{C}$ with both external and internal transitions available.

Example 4. Consider the IMC $\mathcal{C}$ on the right (for instance with a trivial specification not restricting the environment). Note that there are both internal and external actions available in no.



As τ transitions take zero time, the environment $\mathcal{E}$ must spend almost all the time in states without τ. Hence, when ? is entered, $\mathcal{E}$ is almost surely in such a state e. Now τ form ? is taken and $\mathcal{E}$ cannot move to another state when yes/no is entered. Since action a either *is* or *is not* available in e, the environment cannot choose to synchronize in no and not to synchronize in yes. As a result, the environment *"commits" in advance* to synchronize over a either in both yes and no or in none of them. Therefore, in the game we define, **env** cannot completely freely choose which external transition is/is not taken. Further, note that the scheduler of $\mathcal{C}$ cannot observe whether a is currently available in $\mathcal{E}$, which intrinsically induces imperfect information.

In order to transfer these "commitments" to the game, we again make use of the compositionality of IMC and put the product $\mathcal{C} \times \mathcal{S}_i$ in parallel with an IMC *Commit* and then define the game on the result.



The action alphabet of *Commit* is $\mathbb{A}\mathrm{ct} \cup \{\mathbf{Now}, \mathbf{Change}\}$ and the state space is $2^{\mathbb{A}\mathrm{ct}} \cup \{commit, now?\}$ (in the figure, $\mathbb{A}\mathrm{ct} = \{\mathsf{a}\}$; for formal description, see [HKK13]). State $A \subseteq \mathbb{A}\mathrm{ct}$ corresponds to $\mathcal{E}$ being committed to the set of currently available actions A. Thus $A \overset{a}{\hookrightarrow} commit$ for each $a \in A$. This commitment must be respected until the state of $\mathcal{E}$ is changed: either (1) by an external transition from the commitment set (which in *Commit* leads to the state *commit* where a new commitment is immediately chosen); or (2) by a **Change** transition (indicating the environment changed its state due to its Markovian transition).

The game $\mathcal{G}_i$ is played on the arena $\big(\mathcal{C} \times \mathcal{S}_i \parallel_{\mathbb{A}\mathrm{ct}\cup\{\mathbf{Now}\}} Commit\big) \setminus \big(\mathbb{A}\mathrm{ct} \cup \{\mathbf{Now}\}\big)$ with its set of states denoted by G_i. Observe that external actions have either been hidden (whenever they were available in the commitment), or discarded (whenever not present in the current commitment). The only external action that remains is **Change**. The game $\mathcal{G}_i$ is played as follows. There are two types of states: *immediate* states with some τ transitions available and *timed* states with no τ available. The game starts in $v_0 = (c_0, q_0, commit)$.

- In an immediate state $v_n = (c, q, e)$, **con** chooses a probability distribution over transitions corresponding to the internal transitions in $\mathcal{C}$ (if there are any). Then, **env** either approves this choice (chooses ✓) and v_{n+1} is chosen randomly according to this distribution, or rejects this choice and chooses a τ transition to some v_{n+1} such that the transition does *not* correspond to any internal transitions of $\mathcal{C}$. Then the game moves at time $t_{n+1} = t_n$ to v_{n+1}.
- In a timed state $v_n = (c, q, e)$, **env** chooses a delay $d > 0$. Then Markovian transitions (if available) are resolved by randomly sampling a time t according to the exponential distribution with rate $\mathrm{R}(v_n)$ and randomly choosing a target state v_{n+1} where each $v_n \overset{\lambda}{\rightsquigarrow} v$ is chosen with probability $\lambda/\mathrm{R}(v_n)$.
  - If $t < d$, $\mathcal{G}_i$ moves at time $t_{n+1} = t_n + t$ to v_{n+1}, (Markovian transition wins)
  - else $\mathcal{G}_i$ moves at time $t_{n+1} = t_n + d$ to $(c, q, now?)$. ($\mathcal{E}$ takes **Change**)

This generates a run $v_0 t_1 v_1 t_1 \cdots$. The set $(G_i \times \mathbb{R}_{\geq 0})^* \times G_i$ of prefixes of runs is denoted $\mathbb{H}\mathrm{istories}(\mathcal{G})$. We formalize the choice of **con** as a *strategy* $\sigma : \mathbb{H}\mathrm{istories}(\mathcal{G}_i) \to \mathcal{D}(G_i)$. We further allow the **env** to randomize and thus his *strategy* is $\pi : \mathbb{H}\mathrm{istories}(\mathcal{G}_i) \to \mathcal{D}(\{\checkmark\} \cup G_i) \cup \mathcal{D}(\mathbb{R}_{>0})$. We denote by Σ and Π the sets of all strategies of the players **con** and **env**, respectively.

Since **con** is not supposed to observe the state of the specification and the state of *Commit*, we consider in Σ only those strategies that satisfy $\sigma(p) = \sigma(p')$, whenever *observations* of p and p' are the same. Like before, the observation of $(c_0, q_0, e_0)t_1 \cdots t_n(c_n, q_n, e_n) \in \mathbb{H}\mathrm{istories}(\mathcal{G})$ is a sequence obtained from $c_0 t_1 \cdots t_n c_n$ by replacing each maximal consecutive sequence $t_i\, c_i \cdots t_j\, c_j$ with all c_k the same, by $t_i\, c_i$. This replacement takes place so that the player cannot observe transitions that do not affect $\mathcal{C}$. Notice that now $\mathfrak{S}(\mathcal{C})$ is in one-to-one

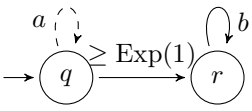
correspondence with Σ. Further, in order to keep CE games out of Zeno behaviour, we consider in Π only those strategies for which the induced Zeno runs have zero measure, i.e. the sum of the chosen delays diverges almost surely no matter what **con** is doing. The *value of* $\mathcal{G}_i$ is now defined as

$$v_{\mathcal{G}_i} \quad := \quad \sup_{\sigma \in \Sigma} \inf_{\pi \in \Pi} \mathcal{P}^{\sigma,\pi}_{\mathcal{G}_i}\left[\Diamond^{\leq T} G\right]$$

where $\mathcal{P}^{\sigma,\pi}_{\mathcal{G}_i}\left[\Diamond^{\leq T} G\right]$ is the probability of all runs of $\mathcal{G}_i$ induced by σ and π and reaching a state with the first component in G before time T. We now show that it coincides with the value of the ith product:

Theorem 2. *For every IMC $\mathcal{C}$, MCA $\mathcal{S}$, $i \in \mathbb{N}$, we have $v_{\mathcal{G}_i} = v_{product}(\mathcal{C} \times \mathcal{S}_i)$.*

This result allows for approximating $v_{\mathcal{S}}(\mathcal{C})$ through computing $v_{\mathcal{G}_i}$'s. However, from the algorithmic point of view, we would prefer approximating $v_{\mathcal{S}}(\mathcal{C})$ by solving a single game $\mathcal{G}$ whose value $v_{\mathcal{G}}$ we could approximate directly. This is indeed possible. But first, we need to clarify, why the approximation sequence $\mathcal{S}_i$ was crucial even in the case where all distributions of $\mathcal{S}$ are already exponential.



Consider the MCA on the right and a conforming environment $\mathcal{E}$, in which a is available iff b becomes available within 0.3 time units. If Player **env** wants to simulate this behaviour, he needs to know how long the transition to r is going to take so that he can plan his behaviour freely, only sticking to satisfying the specification. If we translate Exp(1) directly to a single Markovian transition (with no error incurred), **env** knows nothing about this time as exponential distributions are memoryless. On the other hand, with finer hyper-Erlang, he knows how long the current branch of hyper-Erlang is roughly going to take. In the limit, he knows the precise waiting time right after coming to q.

To summarize, **env** is too weak in $\mathcal{G}_i$, because it lacks the information about the precise time progress of the specification. The environment needs to know how much time is left before changing the location of $\mathcal{S}$. Therefore, the game $\mathcal{G}$ is constructed from $\mathcal{G}_1$ by multiplying the state space with $\mathbb{R}_{\geq 0}$ where we store the exact time to be waited. After the product changes the state so that the specification component switches to a state with $\bowtie d$ constraint, this last component is overwritten with a number generated according to d. This way, the environment knows precisely how much time is left in the current specification location. This corresponds to the infinitely precise hyper-Erlang, where we at the beginning randomly enter a particular branch, which is left in time with Dirac distribution. For more details, see [HKK13].

Denoting the *value of* $\mathcal{G}$ by $v_{\mathcal{G}} := \sup_{\sigma \in \Sigma} \inf_{\pi \in \Pi} \mathcal{P}^{\sigma,\pi}_{\mathcal{G}}\left[\Diamond^{\leq T} G\right]$, we obtain:

Theorem 3. *For every IMC $\mathcal{C}$ and MCA $\mathcal{S}$, we have $v_{\mathcal{G}} = \lim_{i \to \infty} v_{\mathcal{G}_i}$.*

6 Approximation Using Discrete-Time PO Games

In this section, we briefly discuss the approximation of $v_{\mathcal{G}}$ by a discrete time turn-based partial-observation stochastic game Δ. The construction is rather

standard; hence, we do not treat the technical difficulties in great detail (see [HKK13]). We divide the time bound T into N intervals of length $\kappa = T/N$ such that the clock resolution δ (see Section 3.2) satisfies $\delta = n\kappa$ for some $n \in \mathbb{N}$.

1. We enhance the state space with a counter $i \in \{0, \ldots, N\}$ that tracks that $i \cdot \kappa$ time has already elapsed. Similarly, the $\mathbb{R}_{\geq 0}$-component of the state space is discretized to κ-multiples. In timed states, time is assumed to pass exactly by κ. In immediate states, actions are assumed to take zero time.
2. We let at most one Markovian transition occur in one step in a timed state.
3. We unfold the game into a tree until on each branch a timed state with $i = N$ is reached. Thereafter, Δ stops. We obtain a graph of size bounded by $b^{\leq N \cdot |G|}$ where b is the maximal branching and G is the state space of $\mathcal{G}$.

Let Σ_Δ and Π_Δ denote the set of randomized history-dependent strategies of **con** and **env**, respectively, where player **con** observes in the history only the first components of the states, i.e. the states of $\mathcal{C}$, and the elapsed time $\lfloor i/n \rfloor$ up to the precision δ. Then $v_\Delta := \sup_{\sigma \in \Sigma_\Delta} \inf_{\pi \in \Pi_\Delta} \mathcal{P}_\Delta^{\sigma,\pi}(\Diamond G)$ denotes the value of the game Δ where $\mathcal{P}_\Delta^{\sigma,\pi}(\Diamond G)$ is the probability of the runs of Δ induced by σ and π and reaching a state with first component in G. Let b be a constant bounding (a) the sum of outgoing rates for any state of $\mathcal{C}$, and (b) densities and their first derivative for any distribution in $\mathcal{S}$.

Theorem 4. *For every IMC $\mathcal{C}$ and MCA $\mathcal{S}$, $v_\mathcal{G}$ is approximated by v_Δ:*

$$|v_\mathcal{G} - v_\Delta| \quad \leq \quad 10\kappa(bT)^2 \ln \tfrac{1}{\kappa}.$$

A strategy σ^* optimal in Δ defines a strategy $(10\kappa(bT)^2 \ln \frac{1}{\kappa})$-optimal in $\mathcal{G}$. Further, v_Δ and σ^* can be computed in time polynomial in $|\Delta|$, hence in time $2^{\mathcal{O}(|\mathcal{G}|)}$.

The proof of the error bound extends the technique of the previous bounds of [ZN10] and [BHK+12]. Its technical difficulty stems from partial observation and from semi-Markov behaviour caused by the arbitrary distributions in the specification. The game is unfolded into a tree in order to use the result of [KMvS94]. Without the unfolding, the best known (naive) solution would be a reduction to the theory of reals, yielding an EXPSPACE algorithm.

7 Summary

We have introduced an assume-guarantee framework for IMC. We have considered the problem to approximate the guarantee on time-bounded reachability properties in an unknown environment $\mathcal{E}$ that satisfies a given assumption. The assumptions are expressed in a new formalism, which introduces continuous time constraints. The algorithmic solution results from Theorems 1 to 4:

Corollary 1. *For every IMC $\mathcal{C}$ and MCA $\mathcal{S}$ and $\varepsilon > 0$, a value v and a scheduler σ can be computed in exponential time such that $|v_\mathcal{S}(\mathcal{C}) - v| \leq \varepsilon$ and σ is ε-optimal in $v_\mathcal{S}(\mathcal{C})$.*

In future work, we want to focus on identifying structural subclasses of IMC allowing for polynomial analysis.

Acknowledgement. We thank Tomáš Brázdil and Vojtěch Řehák for fruitful discussions and for their feedback.

References

[AH96] Alur, R., Henzinger, T.A.: Reactive modules. In: LICS, pp. 207–218 (1996)

[BF09] Bouyer, P., Forejt, V.: Reachability in stochastic timed games. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part II. LNCS, vol. 5556, pp. 103–114. Springer, Heidelberg (2009)

[BHK+12] Brázdil, T., Hermanns, H., Krčál, J., Křetínský, J., Řehák, V.: Verification of open interactive markov chains. In: FSTTCS, pp. 474–485 (2012)

[BHKH05] Baier, C., Hermanns, H., Katoen, J.-P., Haverkort, B.R.: Efficient computation of time-bounded reachability probabilities in uniform continuous-time Markov decision processes. Theor. Comp. Sci. 345(1), 2–26 (2005)

[BS11] Buchholz, P., Schulz, I.: Numerical Analysis of Continuous Time Markov Decision processes over Finite Horizons. Computers and Operations Research 38, 651–659 (2011)

[CD10] Chatterjee, K., Doyen, L.: The complexity of partial-observation parity games. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 1–14. Springer, Heidelberg (2010)

[DLL+12] David, A., Larsen, K.G., Legay, A., Møller, M.H., Nyman, U., Ravn, A.P., Skou, A., Wasowski, A.: Compositional verification of real-time systems using ECDAR. STTT 14(6), 703–720 (2012)

[EKN+12] Esteve, M.-A., Katoen, J.-P., Nguyen, V.Y., Postma, B., Yushtein, Y.: Formal correctness, safety, dependability and performance analysis of a satellite. In: Proc. of ICSE. ACM and IEEE Press (2012)

[GHKN12] Guck, D., Han, T., Katoen, J.-P., Neuhäußer, M.R.: Quantitative timed analysis of interactive markov chains. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 8–23. Springer, Heidelberg (2012)

[HH13] Hatefi, H., Hermanns, H.: Improving time bounded reachability computations in interactive Markov chains. In: Arbab, F., Sirjani, M. (eds.) FSEN 2013. LNCS, vol. 8161. Springer, Heidelberg (2013)

[HK09] Hermanns, H., Katoen, J.-P.: The how and why of interactive Markov chains. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 311–338. Springer, Heidelberg (2010)

[HKK13] Hermanns, H., Krčál, J., Křetínský, J.: Compositional verification and optimization of interactive markov chains. CoRR, abs/1305.7332 (2013)

[HKR+10] Haverkort, B.R., Kuntz, M., Remke, A., Roolvink, S., Stoelinga, M.I.A.: Evaluating repair strategies for a water-treatment facility using Arcade. In: Proc. of DSN, pp. 419–424 (2010)

[HMP01] Henzinger, T.A., Minea, M., Prabhu, V.S.: Assume-guarantee reasoning for hierarchical hybrid systems. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 275–290. Springer, Heidelberg (2001)

[HNP+11] Hahn, E.M., Norman, G., Parker, D., Wachter, B., Zhang, L.: Game-based abstraction and controller synthesis for probabilistic hybrid systems. In: QEST, pp. 69–78 (2011)

[KKLW07] Katoen, J.-P., Klink, D., Leucker, M., Wolf, V.: Three-valued abstraction for continuous-time Markov chains. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 311–324. Springer, Heidelberg (2007)

[KKN09] Katoen, J.-P., Klink, D., Neuhäußer, M.R.: Compositional abstraction for stochastic systems. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 195–211. Springer, Heidelberg (2009)

[KMvS94] Koller, D., Megiddo, N., von Stengel, B.: Fast algorithms for finding randomized strategies in game trees. In: STOC, pp. 750–759 (1994)

[KV96] Kupferman, O., Vardi, M.: Module checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 75–86. Springer, Heidelberg (1996)

[KZH+11] Katoen, J.-P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. Performance Evaluation 68(2), 90–104 (2011)

[LT88] Larsen, K.G., Thomsen, B.: A modal process logic. In: LICS, pp. 203–210 (1988)

[Mar11] Markovski, J.: Towards supervisory control of interactive Markov chains: Controllability. In: ACSD, pp. 108–117 (2011)

[MC81] Misra, J., Mani Chandy, K.: Proofs of networks of processes. IEEE Trans. Software Eng. 7(4), 417–426 (1981)

[RW89] Ramadge, P.J.G., Wonham, W.M.: The control of discrete event systems. Proceedings of the IEEE 77(1) (1989)

[Spr11] Sproston, J.: Discrete-time verification and control for probabilistic rectangular hybrid automata. In: QEST, pp. 79–88 (2011)

[TAKB96] Tasiran, S., Alur, R., Kurshan, R.P., Brayton, R.K.: Verifying abstractions of timed systems. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 546–562. Springer, Heidelberg (1996)

[ZN10] Zhang, L., Neuhäußer, M.R.: Model checking interactive Markov chains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 53–68. Springer, Heidelberg (2010)

Thermodynamic Graph-Rewriting

Vincent Danos[1], Russ Harmer[2], and Ricardo Honorato-Zimmer[1]

[1] School of Informatics, University of Edinburgh
[2] CNRS & Université Paris-Diderot

Abstract. We develop a new 'thermodynamic' approach to stochastic graph-rewriting. The ingredients are a finite set of reversible graph-rewriting rules $\mathcal{G}$ (called generating rules), a finite set of connected graphs $\mathcal{P}$ (called energy patterns), and an energy cost function $\epsilon : \mathcal{P} \to \mathbb{R}$. The idea is that $\mathcal{G}$ defines the qualitative dynamics by showing which transformations are possible, while $\mathcal{P}$ and ϵ specify the long-term probability π of any graph reachable under $\mathcal{G}$. Given $\mathcal{G}$, $\mathcal{P}$, we construct a finite set of rules $\mathcal{G}_{\mathcal{P}}$ which (i) has the same qualitative transition system as $\mathcal{G}$, and (ii) when equipped with suitable rates, defines a continuous-time Markov chain of which π is the unique fixed point. The construction relies on the use of site graphs and a technique of 'growth policy' for quantitative rule refinement which is of independent interest. The 'division of labour' between the qualitative and the long-term quantitative aspects of the dynamics leads to intuitive and concise descriptions for realistic models (see the example in §4). It also guarantees thermodynamical consistency (*aka* detailed balance), otherwise known to be undecidable, which is important for some applications. Finally, it leads to parsimonious parameterizations of models, again an important point in some applications.

1 Introduction

Along with Petri nets, communicating finite state machines, and process algebras, models of concurrent systems based on graphs and graph transformations (GTS) have long been investigated as means to describe, verify and synthesize distributed systems [11]. Beyond their visual aspect, which is useful in modelling situations, there is a lot to like about GTSs: there are category-theoretic frameworks to express them and encapsulate their syntax; and the existence of a strong meta-theory [19] is a reassurance that methodologies developed in specific cases can be 'ported' to other variants.

Graph-rewriting rules are convenient for writing compact models and modifying them [7], and lend themselves naturally to probabilistic extensions [16,18]. However, for all their flexibility, even rules can only do so much. We ask in this paper "what if we did not have to write the rules?". This is where we take a page from the book of classical statistical mechanics. In such models, which often involve graph-like structures as in the Ising model, the dynamics is not described upfront. Instead, the system of interest is equipped with an 'energy

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 380–394, 2013.

lansdcape' which specifies its long run behaviour, be it deterministic as in classical mechanics, or probabilistic in statistical physics. The dynamics just follows from the energy data. In the eye of a computer scientist, this use of energy looks like a latent syntax. (This is especially true in the application of these ideas to molecular dynamics.)

The broad aim of this paper is to make this syntax explicit by introducing energy patterns and costs from which the total energy of a state of the system can be computed; and to define a procedure whereby, indeed, the dynamics described as probabilistic graph-rewriting rules can be derived from these energy data. Descriptively, this takes us to an entirely new level of conciseness (as in the example §4). It also guarantees thermodynamical consistency, otherwise known to be undecidable [8], which is important for some applications. But perhaps the nicest byproduct of this approach is the fact that the methodology leads to parsimonious parameterizations. The parameter space which usually scales as the number of rules (which in turn has at best a logarithmic impact on the cost of a simulation event [4]), will now scale as the number of energy patterns provided in the specification.

The particular kind of GTSs we consider forms a reversible subset of the Kappa site-graph stochastic rewriting language. Kappa is used for the simulation and analysis of combinatorial dynamical systems as typically found in cellular signalling networks [20,24] and has been predicted to "become one of the mainstream modelling tools of systems biology within the coming decade" [1]. Similar graph formalisms where nodes have a controlled valence/degree have been considered *e.g.* the BNG language [12,17], Kissinger and Dixon's quantum proof language [10], and Kirchner *et al.* chemical calculi [3]. Site-graph rewriting has found recently a 'home' both in the single-pushout GTS tradition [5] and the double-pushout one [14]. This makes one hopeful that the thermodynamic methodology we propose can crossover to other fields where quantitative GTSs can be used, *e.g.* in the modelling of adaptive networks [13]. While our scalable energy-based parameterization is particularly important in biological applications where parameters often need to be inferred, one can imagine it to be useful in other modelling situations with uncertainty.

Outline: We start with the definition and relevant properties of the specific GTS we use, namely a simple reversible fragment of Kappa. Next, we introduce growth policies (adapted from Ref. [23]), a tool which allows one to replace a rule with an orthogonal set of refined rules while preserving the quantitative semantics. We use this tool with a specific policy which refines a rule into finitely many rules, each of which has a definite energy balance with respect to a given set of energy patterns. This leads to our main theorem which guarantees that the stochastic dynamics of the obtained refined rule set converges to an equilibrium distribution parametrized by the cost of each energy pattern. Throughout, the presentation is set in category-theoretic terms and mostly self-contained. A substantial example concludes the paper. (For lack of space, and following the advice of the referees, proofs were omitted in this extended abstract; these will be presented in a longer version.)

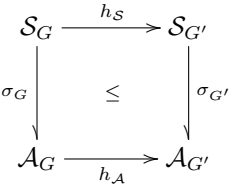
2 Site Graph Rewriting

2.1 Site Graphs and Homomorphisms

A *site graph* G consists of a finite sets of *agents* and *sites*, $\mathcal{A}_G$ and $\mathcal{S}_G$, a partial function $\sigma_G : \mathcal{S}_G \rightharpoonup \mathcal{A}_G$, and a symmetric *edge* relation $\mathcal{E}_G$ on $\mathcal{S}_G$. The pair $\mathcal{A}_G$, $\mathcal{E}_G$ form an undirected graph; sites not in the domain of $\mathcal{E}_G$ are said to be *free*. The role of the additional map σ_G is to assign sites to agents; sites not in the domain of σ_G are said to be *dangling*, and will be used to represent half-edges (see below). Usually one also endows agents and/or sites with states (see §4); the construction we will give in §3 carries over trivially to these.

One says G is *realizable* iff (i) no site has an edge to itself; (ii) sites have at most one incident edge; (iii) no dangling site is free; and, (iv) edges have at most one dangling site.

A *homomorphism* $h : G \to G'$ of site graphs is a pair of functions, $h_{\mathcal{S}} : \mathcal{S}_G \to \mathcal{S}_{G'}$ and $h_{\mathcal{A}} : \mathcal{A}_G \to \mathcal{A}_{G'}$, such that (i) whenever $h_{\mathcal{A}}(\sigma_G(s))$ is defined, so is $\sigma_{G'}(h_{\mathcal{S}}(s))$ and they are equal; and (ii) if $s \, \mathcal{E}_G \, s'$ then $h_{\mathcal{S}}(s) \, \mathcal{E}_{G'} \, h_{\mathcal{S}}(s')$.



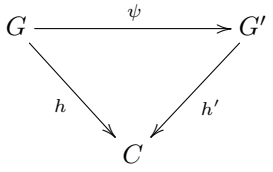
A homomorphism $h : G \to G'$ is an *embedding* iff (i) $h_{\mathcal{A}}$ and $h_{\mathcal{S}}$ are injective; and (ii) if s is free in G, so is $h_{\mathcal{S}}(s)$ in G'. If $h : G \to G'$ is an embedding and G' is realizable then G is also realizable.

Site graphs and homomorphisms form a category **SG** with the natural 'tiered' composition; embeddings form a subcategory; if in addition, we restrict objects to be realizable, we get the subcategory **rSGe** of realizable site graphs and embeddings.

2.2 The Category of Site Graphs over C

A homomorphism $h : G \to C$ is a *contact map* over C iff (i) G is realizable, (ii) σ_C is total and (iii) whenever $h_{\mathcal{S}}(s_1) = h_{\mathcal{S}}(s_2)$ and $\sigma_G(s_1) = \sigma_G(s_2)$, then $s_1 = s_2$. The third condition of local injectivity means that every agent of G has at most one copy of each site of its corresponding agent in C; C is called the *contact graph*.

Hereafter, we work in the (comma) category $\mathbf{rSGe}_C$ whose objects are contact maps over C, and arrows are embeddings such that the associated triangle commute in **SG**. We write $\Upsilon(h, h')$ for the set of such embeddings between h, h' contact maps over C; we also write $|\_|$ for the domain functor from $\mathbf{rSGe}_C$ to **rSGe** which forgets types. In particular, if $h : G \to C$ is a contact map, we write $|h|$ for its source G.



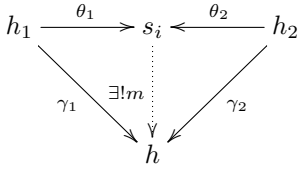
The contact graph C is fixed and plays the role of a *type*: it specifies the kinds of agents that exist, the sites that they may possess, and which of the $|\mathcal{S}_C|^2$

possible edge types are actually valid. It also gives canonical names to the types of agents and their sites. In examples, we write agent and sites names directly.

In **rSGe**, a dangling site s in G can be mapped to any site of G' by an embedding $h : G \to G'$; as such, it can be used as an *any site* wild card when matching G. In $\mathbf{rSGe}_C$, the contact map $c : G \to C$ tells us which agent A in C the site s belongs to because σ_C is total, and this must be respected by h; we call this a *binding type* wild card since it typically allows us to express the property of being bound to the site s of *some occurrence* of the agent A.

The category **SG** has all pull-backs, constructed from those in **Set**; it is easy to see that they restrict to $\mathbf{rSGe}_C$. The category **SG** also has sums, but these do not restrict to $\mathbf{rSGe}_C$. (Just like sums in **Set** do not restrict to the subcategory of injective maps.)

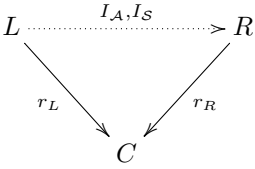
However, $\mathbf{rSGe}_C$ has *multi-sums*: meaning for all pairs of site graphs of type C, $h_1 : G_1 \to C$ and $h_2 : G_2 \to C$, there exists a family of co-spans $\theta_1^i : h_1 \to s_i \leftarrow h_2 : \theta_2^i$, such that any co-span $\gamma_1 : h_1 \to h \leftarrow h_2 : \gamma_2$ factors through *exactly one* of the family and does so *uniquely*. The idea is that the pairs θ_1^i, θ_2^i enumerate all minimal ways in which one can glue h_1 and h_2, that is to say all the minimal glueings of G_1 and G_2 that respect C. There are *finitely* many which all factor through the standard sum in the larger slice category $\mathbf{SG}_C$.



The notion of multi-sum dates back to Ref. [9]; we will call them *minimal glueings* in **rSGe** according to their intuition in this concrete context, and use them in §3.2 to construct balanced rules.

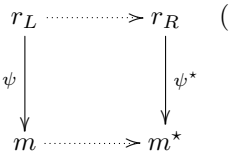
2.3 Rules

A *rule* r over C is a pair of contact maps $r_L : L \to C$, $r_R : R \to C$ which differ only in their edge structures, *i.e.* $\mathcal{A}_L = \mathcal{A}_R$, $\mathcal{S}_L = \mathcal{S}_R$, $\sigma_L = \sigma_R$, $r_{L\mathcal{A}} = r_{R\mathcal{A}}$, and $r_{L\mathcal{S}} = r_{R\mathcal{S}}$.



A contact map $m : M \to C$ is a *mixture* iff σ_M is total (no dangling edge) and, for all $a \in \mathcal{A}_M$, $\sigma_M^{-1}(a) \cong \sigma_C^{-1}(m_{\mathcal{A}}(a))$, *i.e.* $m_{\mathcal{S}}$ is locally surjective. In words, a mixture is a *fully-specified* site graph with respect to the type C.

An embedding $\psi : r_L \to m$ induces a *rewrite* of a mixture m by modifying the edge structure of the image of ψ, *i.e.* an instance of L in M, to that of R; the result of rewriting is a new mixture $m^\star$, where $|m^\star|$ has the same agents and sites as $M = |m|$, and an embedding $\psi^\star : r_R \to m^\star$. This can be formalized using double push-out rewriting [5] (since all the required push-outs do exist in $\mathbf{rSGe}_C$). But with the simple rules considered here, there is no need.



(1)

We also write $\Upsilon(r, m)$ for the set of all embeddings $\psi : r_L \to m$.

The inverse of r, defined as $r^\star := (r_R, r_L)$ is also a valid rule; by rewriting $m^\star$ with $r^\star$ via $\psi^\star$, we recover m and ψ.

Given a finite set of rules $\mathcal{G}$ over C, we define a labelled transition system $\mathcal{L}_\mathcal{G}$ on mixtures over C: a transition from a mixture m is a rewriting step determined and labelled by an 'event' (r, ψ) as in diagram (1); with r in $\mathcal{G}$, and ψ in $\Upsilon(r, m)$.

We suppose hereafter that $\mathcal{G}$ is closed under rule inversion, *i.e.* $\mathcal{G} = \mathcal{G}^\star$. Hence, every (r, ψ)-transition has an inverse $(r^\star, \psi^\star)$, and $\mathcal{L}_\mathcal{G}$ is symmetric.

2.4 CTMC Semantics

It is not difficult to see that for any rule r, $|\Upsilon(r, m)| \leq |\mathcal{A}_{|m|}|^{d(r)}$ where $d(r)$ is the number of connected components in r_L. Hence, $\mathcal{L}_\mathcal{G}$ has finite out-degree, bounded by $|\mathcal{G}| \cdot |\mathcal{A}_{|m|}|^d$ for some d. Also, as agents are preserved by rules, the (strongly) connected components of $\mathcal{L}_\mathcal{G}$ are finite.

Hence, given a *rate map* k from $\mathcal{G}$ to $\mathbb{R}_+$, we can equip $\mathcal{L}_\mathcal{G}$ with the structure of an irreducible continuous-time Markov chain (CTMC), simply by assigning rate $k(r)$ to an event of the form (r, ψ).

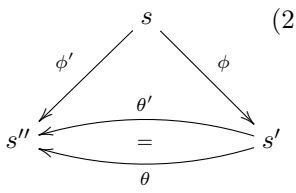
We write $\mathcal{L}^k_\mathcal{G}$ for the obtained CTMC.

We need here to record a definition for later use: a finite CTMC $\mathcal{M}$ has *detailed balance* for a probability distribution π on $\mathcal{M}$'s state space, if for all states x and y, $\pi(x) \cdot q(x, y) = \pi(y) \cdot q(y, x)$ where $q(x, y)$ is $\mathcal{M}$'s transition rate from x to y. This implies that, assuming $\mathcal{M}$ is irreducible, π is the unique fixed point of the action of $\mathcal{M}$, to which the probabilistic state of $\mathcal{M}$ converge, regardless of the initial state.

2.5 Extensions and Rule Refinement

Epis of $\mathbf{rSGe}_C$ can be characterized as follows [23]: suppose $s : G \to C$ and $s' : G' \to C$ are contact maps then $\phi : s \to s'$ of $\mathbf{rSGe}_C$ is an epi iff every connected component of G' contains at least one agent in the image of $\phi_\mathcal{A}$.

We refer to an epi $\phi : s \to s'$ as an *extension* of s. The category of extensions of s is a pre-order, *i.e.* there is at most one arrow between any two objects: if $\phi' = \theta\phi = \theta'\phi$ then $\theta = \theta'$ because ϕ is an epi. We write $\phi \leq \phi'$ for this *specialization* order. If $\phi \leq \phi'$ and $\phi' \leq \phi$ then we write $\phi \cong_s \phi'$.



(2)

A family of epis $\phi_i : s \to t_i$ *uniquely decomposes* s iff, for all mixtures m and embeddings $h : s \to m$, there exists a unique i and ψ such that $h = \psi\phi_i$. This is the basic requirement for a reasonable notion of rule refinement: it guarantees that the LHS s of a given rule splits into a non-overlapping collection of more specific cases t_i.

In the next Section, we will be constructing specific such decompositions in order to produce families of sub-rules which are compatible with energy patterns.

First, we recall the growth policy method to find such unique decompositions which works by detailing which agents and sites should be added to s.

Specifically, a *growth policy* Γ for s is a family of functions Γ_ϕ, indexed by extensions $\phi : s \to t$, where Γ_ϕ maps $u \in \mathcal{A}_{|t|}$ to a subset $\Gamma_\phi(u)$ of $\sigma_C^{-1}(t_\mathcal{A}(u))$, *i.e.* each agent in $|t|$ is allocated a subset of the sites its sites can map to in C. An agent in $|t|$ may cover some, or all, of these sites or even completely extraneous sites: if the former, *i.e.* for all u in $\mathcal{A}_{|t|}$, $t_\mathcal{S}(\sigma_{|t|}^{-1}(u)) \subseteq \Gamma_\phi(u)$, we say that ϕ is *immature*; if for all us, the inclusion is an equality, we say that ϕ is *mature*; otherwise ϕ is said to be *overgrown*. The functions Γ_ϕ must satisfy, for all extensions ϕ and $\phi' \geq \phi$, the *faithfulness* property, $\Gamma_\phi = \Gamma_{\phi'}\psi_\mathcal{A}$, where ψ is the epi witnessing $\phi \leq \phi'$; so a site requested by ϕ must be requested by any further extension. If ϕ is not overgrown then no $\phi' \leq \phi$ is overgrown either. Also, note that the *union* of two growth policies is itself a growth policy.

Given an s and a growth policy Γ for s, we define $\Gamma(s)$ by choosing one representative per $\cong_s$-isomorphism class of the set of all extensions of s which are mature according to Γ.

Theorem 1. *If Γ is a growth policy for s, then $\Gamma(s)$ uniquely decomposes s.*

The theorem (adjusted from Ref. [23]) guarantees that factorizations through $\Gamma(s)$ are unique, but not that they always exist. In the next section, we will construct a growth policy for which this property of exhaustivity of the decomposition can be proved by hand.

Given a rule r and an extension ϕ of r_L, r_ϕ denotes the 'refined' rule associated to ϕ. If Γ is a growth policy for r_L, the *refinement* of r by Γ is the set of rules, $\Gamma(r)$, the elements of which are of the form r_ϕ, for ϕ in $\Gamma(r_L)$ a mature extension.

It is easy to see that due to the simple nature of our rules, the category of extensions of r_L and r_R are isomorphic; if ϕ is an extension of r_L, we will write $\phi^\star$ for the corresponding extension of r_R.

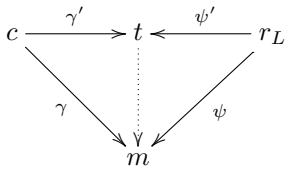
An example of growth policy is the *ground* policy which assigns all possible sites to all agents. In which case: $\Gamma(s)$ is simply the set, possibly infinite, of epis of s into mixtures, considered up to $\cong_s$; and $\Gamma(r)$, the ground refinement of r, contains all refinements of r along these epis, which therefore directly manipulate mixtures.

3 Rule Generation

We fix a finite set $\mathcal{G}$ of *generator* rules; and a finite set $\mathcal{P}$ of connected contact maps in $\mathbf{rSGe}_C$; these are our *energy patterns*.

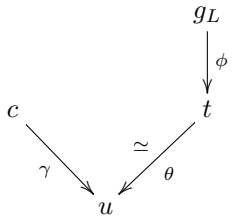
The goal is now to refine $\mathcal{G}$ into a new rule set $\mathcal{G}_\mathcal{P}$ where each refined rule is $\mathcal{P}$-*balanced*, which means that, however applied, it consumes or produces a fixed amount of each pattern c in $\mathcal{P}$. The construction proceeds in two steps: first, we characterize balanced refinements; second, we define a growth policy with balanced mature extensions, and apply Th. 1. Note that ground extensions of g are trivially balanced but, in general, the ground refinement is impractically large or even infinite; ours will always be finite.

3.1 $\mathcal{P}$-Balanced Extensions



Consider c in $\mathcal{P}$, and a rule r. For an r-event ψ to *consume* an instance γ of c in a mixture m, the cospan $(\gamma_{\mathcal{S}}, \psi_{\mathcal{S}})$ must have images which intersect on at least one edge modified by r. This is the case iff the associated minimal glueing (γ', ψ') —obtained by restricting the cospan to the union of its images in m— has the same property. Likewise, for an r-event to *produce* an instance of c, the associated minimal glueing between c and r_R must have a modified intersection. We call such minimal glueings *relevant*; they are the ones which underlie events that can affect the set of instances of c.

Pick g in $\mathcal{G}$ and $\phi : g_L \to t$ an extension of g_L. One says that ϕ is $\mathcal{P}$-*left-balanced* iff, for all relevant minimal glueings $\gamma : c \to u \leftarrow t : \theta$ with $c \in \mathcal{P}$, θ is an isomorphism. This means that the image of c under γ is contained in t. Symmetrically, one says that ϕ is $\mathcal{P}$-*right-balanced* iff $\phi^\star$ is a $\mathcal{P}$-left-balanced extension of $r^\star$.



(3)

An extension ϕ is $\mathcal{P}$-*balanced* iff it is $\mathcal{P}$-left- and $\mathcal{P}$-right-balanced; we say that ϕ is *prime* iff it is minimal $\mathcal{P}$-balanced in the specialization order $\leq$.

If ϕ is a $\mathcal{P}$-balanced extension of g, the refined rule g_ϕ has a *balance vector* in $\mathbb{Z}^{\mathcal{P}}$, written $\Delta\phi$, where $\Delta\phi(c)$, for $c \in \mathcal{P}$, is the amount of c produced by *any* g_ϕ-event leading from m to $m^\star$, or equivalently the difference between the number of embeddings of c in the RHS and the LHS of g_ϕ. Indeed, as ϕ is balanced, $|\Upsilon(c, m^\star)| - |\Upsilon(c, m)| = |\Upsilon(c, g_{\phi,R})| - |\Upsilon(c, g_{\phi,L})|$.

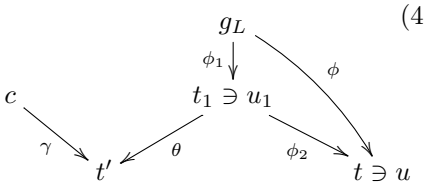
Conversely, a non-$\mathcal{P}$-balanced extension will incur different $\Delta\phi(c)$ for well-chosen applications of g_ϕ, if c in $\mathcal{P}$ violates the condition of diagram (3). Thus, the notion of balanced extension characterizes the property that we want. (This would no longer be the case if one were to relativize the construction to a superset of reachables; *e.g.* in order to reduce the size of the generated rule set.)

3.2 Add-by-Glueing

We now define a growth policy, which uses minimal glueings on non-$\mathcal{P}$-balanced extensions ϕ to add further required sites into ϕ's codomain; this corresponds in diagram (3) to the case where θ is not an isomorphism.

Some care is needed to ensure faithfulness, *i.e.* $\Gamma_\phi = \Gamma_{\phi'\phi}\,\phi'_{\mathcal{A}}$, since relevant minimal glueings on ϕ can disappear along a further extension ϕ' and, consequently, a site that was 'requested' at ϕ may no longer be so after at $\phi'\phi$. To address this, we add site requests from all relevant minimal glueings in the *past* of an extension.

Given $g \in \mathcal{G}$ we define a growth policy Γ_L for g_L. Suppose $\phi : g_L \to t$ is an extension of g_L. We set $\Gamma_L(\phi)$ to request a site s in $\sigma_C^{-1}(t_{\mathcal{A}}(u))$ at agent $u \in \mathcal{A}_{|t|}$ iff either (i) $u = \phi_{\mathcal{A}}(u_0)$ and $s = \phi_{\mathcal{S}}(s_0)$ for some u_0 in $\mathcal{A}_{|g_L|}$, s_0 in $\mathcal{S}_{|g_L|}$; or (ii) ϕ factorizes as $\phi_2\phi_1$, where $\phi_1 : s \to t_1$, and there is a relevant minimal glueing $\gamma : c \to t' \leftarrow t_1 : \theta$, u_1 in $\mathcal{A}_{|t_1|}$, and a site s_1' in $\sigma_{|t'|}^{-1}(\theta_{\mathcal{A}}(u_1))$ such that $u = \phi_{2,\mathcal{A}}(u_1)$, and $s = t'_{\mathcal{S}}(s_1')$. (4)



The first clause simply ensures that all sites already covered in g_L are asked for; the second one adds in sites which appear by glueing at some point between g_L and t. We refer to $\phi_2 : t_1 \to t$ as a *rewind* of ϕ.

Symmetrically, we define a growth policy Γ_R for g_R by applying the same definition to the reverse generator $g^\star$. Since extensions of g_L and g_R are isomorphic, we can, with a slight abuse of notation, define $\Gamma^{\mathcal{P}} := \Gamma_L \cup \Gamma_R$.

Theorem 2. *The above $\Gamma^{\mathcal{P}}$ is indeed a growth policy for g_L; the induced refined rule set $\Gamma^{\mathcal{P}}(g)$ is non-empty, balanced, exhaustive and finite.*

Therefore, given $\mathcal{G}$ and $\mathcal{P}$, we obtain a finite $\mathcal{P}$-balanced rule set which refines $\mathcal{G}$ exhaustively, by setting $\mathcal{G}_{\mathcal{P}} := \dot{\cup}_{g \in \mathcal{G}} \Gamma^{\mathcal{P}}(g)$ (disjoint sum). To every refinement g_ϕ, corresponds an inverse refinement $g^\star_{\phi^\star}$; hence, $\mathcal{G}_{\mathcal{P}} = \mathcal{G}^\star_{\mathcal{P}}$ is closed under inversion like $\mathcal{G}$.

3.3 Rates

To equip $\mathcal{G}_{\mathcal{P}}$ with rates, we suppose given a $\mathcal{P}$-indexed real-valued vector of *energy costs* ϵ, and a rate map $k : \mathcal{G}_{\mathcal{P}} \to \mathbb{R}_+$ such that, for all g_ϕ in $\mathcal{G}_{\mathcal{P}}$:

$$\log k(g^\star_{\phi^\star}) - \log k(g_\phi) = \epsilon \cdot \Delta\phi \tag{5}$$

with $\Delta\phi$ in $\mathbb{Z}^{\mathcal{P}}$, the balance vector of the refined rule g_ϕ with respect to $\mathcal{P}$, a well-defined quantity by Th. 2.

We write $\mathcal{P}(x)$ for the $\mathcal{P}$-indexed vector which maps c to $|\Upsilon(c, x)|$, and define the *energy* $E(x)$ of x as $\epsilon \cdot \mathcal{P}(x)$. We also write $\mathcal{L}_{\mathcal{G}}(x)$ for the finite (strongly) connected component of x in $\mathcal{L}_{\mathcal{G}}$, and define a probability distribution (in Boltzmann format) on $\mathcal{L}_{\mathcal{G}}(x)$ by:

$$\pi_x(y) := e^{-\epsilon \cdot \mathcal{P}(y)} / \sum_{y \in \mathcal{L}_{\mathcal{G}}(x)} e^{-\epsilon \cdot \mathcal{P}(y)} \tag{6}$$

Theorem 3. *Let $\mathcal{G}$, $\mathcal{P}$, $\mathcal{G}_{\mathcal{P}}$, k, and π_x be as above; $\mathcal{L}_{\mathcal{G}_{\mathcal{P}}}$ and $\mathcal{L}_{\mathcal{G}}$ are isomorphic as symmetric LTSs; and, for any mixture x, the irreducible continuous-time Markov chain $\mathcal{L}^k_{\mathcal{G}_{\mathcal{P}}}$ has detailed balance for, and converges to π_x, on $\mathcal{L}_{\mathcal{G}_{\mathcal{P}}}(x) = \mathcal{L}_{\mathcal{G}}(x)$ the finite strongly connected component of x.*

Note that the subset of the state space which is reachable from x in $\mathcal{L}_\mathcal{G}$, namely $\mathcal{L}_\mathcal{G}(x)$ is finite; hence, the *partition function* $Z(x) = \sum_{y \in \mathcal{L}_\mathcal{G}(x)} e^{-E(y)}$ is finite. With rules which increase the number of agents, components $\mathcal{L}_\mathcal{G}(x)$ can be infinite, and $Z(x)$ may diverge. For (mass action stochastic) Petri nets, convergence is guaranteed if detailed balance holds, but it is not true in general for Kappa [8,6].

Another point worth making is that the result holds symbolically—regardless of the energy cost ϵ. So ϵ can be seen as a set of parameters, an ideal support for machine learning techniques if one were contemplating fitting a model to data.

3.4 A Linear Kinetic Model

So, the reader will ask, what of the actual rates of $\mathcal{L}^k_{\mathcal{G}_\mathcal{P}}$? Among all possible choices which accord with (5), it is possible to delineate a tractable subset the size of which grows quadratically in $|\mathcal{P}|$. This is a useful log-linear heuristics, which is common in machine learning but has no claim to validity.

We keep the same notations as in Th. 3.

Suppose we have, for every generating rule g in $\mathcal{G}$, a constant $c_g \in \mathbb{R}$, and a matrix A_g of dimension $|\mathcal{P}| \times |\mathcal{P}|$. Subject to the constraints that $c_{g^\star} = c_g$, and $A_{g^\star} + A_g = I$, we can define a log-affine rate map which satisfies (5) by:

$$\log(k(g_\phi)) := c_g - A_g(\epsilon) \cdot \Delta\phi \tag{7}$$

The kinetic model expressed in (7) requires of the order of $|\mathcal{P}|^2 \times |\mathcal{G}|$ parameters. In practice, one needs even fewer parameters, as only those energy patterns that are relevant to a given g, *i.e.* have a non-zero balance for at least one rule in $\Gamma^\mathcal{P}(g)$, need to be considered when building A_g. Typically, for larger models, this will be a far smaller number than $|\mathcal{P}|$. This relative parsimony is compounded by the fact that the number of *independent* parameters will be often lower, because the $\Delta\phi$ family has often low rank. It is to be compared with the total number of choices possible which is far greater as it is of the order of the number of refinements, that is to say $\sum_{g \in \mathcal{G}} |\Gamma^\mathcal{P}(g)|$.

If we set $c_{g^\star} = c_g = 0$, $A_{g^\star} = 0$, $A_g = I$, we get: $k(g_\phi) = e^{-\epsilon \cdot \Delta\phi}$, $k(g^\star_{\phi^\star}) = 1$. As $\epsilon \cdot \Delta\phi$ is the difference of energy between the target and source in any application g_ϕ, this choice amounts to being exponentially reluctant to climb up the energy gradient. This is a continuous-time version of the celebrated Metropolis algorithm [22].

4 Allosteric Ring

We can put our energy-based modelling methodology to use on a realistic example of a bacterial flagellar engine. In this section, we will prefer the traditional syntax of Kappa to denote site graphs: namely subscripts for states and shared superscripts for edges between sites, *e.g.* $A(x^1_0), B(y^1)$. Differently from the mathematical definitions of §2, agent and site types are indicated as explicit labels.

We use KaSim (`https://github.com/jkrivine/KaSim`), the standard Kappa engine, for the simulation shown below.

The engine can rotate clockwise or anti-clockwise at high angular velocities, and this will decide whether the bacterium tumbles or swims forward. One can build a simple model of the switch between the two modes [2]. The engine is seen as a ring of n identical components, P, with two possible conformations, 0 and 1. (In reality, each of the $n = 34$ component protomers is itself a tiny complex made of different subcomponents, but the model ignores this.) A ring homogeneously in state 0 (1) rotates (anti-) clockwise and induces tumbling (straight motion). Importantly, neighbouring Ps on the ring prefer to have matching conformations. States of the ring with many mismatches thus incur high penalties. In the absence of any Y molecule binding a P, its favoured conformation is 0; conversely, in the presence of a Y, P favours 1. (Y stands for a small diffusible protein named *CheY*.) To bind, Y has to be activated by an external signal. Hence the switch can be triggered by a sudden activation of Y which then binds the ring and induces a change of regime. The sharper the transition between the two regimes the better.

As each of the Ps can be in four states, the ring has on the order of 10^{20} non-isomorphic configurations which precludes any reaction-based (*e.g.* Petri nets) approach to the dynamics where each global state is considered as one chemical species. At this stage, we could apply the rule-based approach, or, better, we can obtain the rules *indirectly* by applying the methodology of §3. This is what we do now informally.

First, we define our contact graph with two agent types: $P(x, y, f_{0,1}, s)$ with domains x, y to form the ring, s to bind its signal Y, and f a placeholder for P's conformation; $Y(s_{u,p})$ with two internal states to represent activity.

Second, we capture the informal statements in the discussion above by defining the energy patterns and associated costs. Note that the various motifs overlap. Following §3, we associate to each ring configuration x the occurrence vector $\mathcal{P}(x)$ and total energy $\epsilon \cdot \mathcal{P}(x)$. For example, a ring of size n uniformly in state 0 and with no bound Ys has total energy $n(\epsilon_{00}^{PP} + \epsilon_0^P)$. This, in turn, defines the equilibrium distribution of the ring, namely x has probability proportional to $\exp(-\epsilon \cdot \mathcal{P}(x))$. (The convention is that the lower the energy, the likelier the state.)

Motif	*Cost*
$P(f_i, x^1), P(y^1, f_j)$	ϵ_{ij}^{PP}
$P(f_i)$	ϵ_i^P
$P(f_i, s^1), Y(s^1)$	ϵ_i^{PY}

In order to complete our energy landscape, we need to pick energy costs which reward or penalize local configurations as discussed above: the role of (8) is to align the internal states of neighbours on the ring — an Ising penalty term for mismatching neighbours which will "spread conformation"; (9) makes 0 the favoured state, while (10), which kicks in only in the presence of Y, makes 1 the favoured state.

$$\epsilon_{00}^{PP}, \epsilon_{11}^{PP} < \epsilon_{10}^{PP}, \epsilon_{01}^{PP} \quad (8)$$

$$\epsilon_0^P < \epsilon_1^P \quad (9)$$

$$\epsilon_0^{PY} > \epsilon_1^{PY} \quad (10)$$

The next step is to create the dynamics. The naive rule b for PY binding:

$$b := P(s), Y(s_p) \leftrightarrow P(s^1), Y(s_p^1)$$

has a ΔE which is ambiguous as it will be either ϵ_0^{PY} or ϵ_1^{PY} depending on its instances; hence, we have no hope of assigning rates to this rule that satisfy detailed balance—unless $\epsilon_0^{PY} = \epsilon_1^{PY}$, which contradicts (10). To get a definite balance, one needs to refine this rule:

$$b_0 := P(f_0, s), Y(s_p) \leftrightarrow P(f_0, s^1), Y(s_p^1)$$
$$b_1 := P(f_1, s), Y(s_p) \leftrightarrow P(f_1, s^1), Y(s_p^1)$$

Now each rule b_i specifies enough of the context in which it applies to have a definite energy balance ϵ_i^{PY}. Following the same intuition of revealing (just) enough context, we obtain a balanced rule set for conformational changes:

$$f_{ij} := P(f_i, y^1), P(x^1, f_0, y^2, s), P(x^2, f_j) \leftrightarrow P(f_i, y^1), P(x^1, f_1, y^2, s), P(x^2, f_j)$$
$$f'_{ij} := P(f_i, y^1), P(x^1, f_0, y^2, s^{\text{-}}), P(x^2, f_j) \leftrightarrow P(f_i, y^1), P(x^1, f_1, y^2, s^{\text{-}}), P(x^2, f_j)$$

The first (second) group of rules represents the changes in the absence (presence) of a Y bound to the middle P undergoing a change of conformation. (The fact that P's site s is bound is indicated by the underscore exponent.)

These f-rules have respective balance:

$$\epsilon_{i1}^{PP} + \epsilon_{1j}^{PP} - \epsilon_{i0}^{PP} - \epsilon_{0j}^{PP} + \epsilon_1^P - \epsilon_0^P$$
$$\epsilon_{i1}^{PP} + \epsilon_{1j}^{PP} - \epsilon_{i0}^{PP} - \epsilon_{0j}^{PP} + \epsilon_1^P - \epsilon_0^P + \epsilon_1^{PY} - \epsilon_0^{PY}$$

As we have ten reversible rules, and only eight energy patterns, there must be linear dependencies between the various balances. Indeed, in this case, it is easy to see that the family of vector balances has rank six. Thermodynamic consistency induces relationships between rates; a well-established fact in the case of reaction networks (*e.g.* see Ref. [6]).

With the rules in place, the final step is to choose rates which satisfy detailed balance. This guarantees that the obtained rule set converges to the equilibrium specified by the choice of the energy cost vector. Convergence will happen whatever ϵ is, ie symbolically. If, in addition, ϵ follows (8–10), one can see in Fig. 1 that the ring 1) undergoes sharp transitions when active Y is stepped up and down again, and 2) has at all times very few mismatches.

4.1 How to Generate the Rules

Our set of balanced rules for the ring dynamics was based on two generators, b for binding, f for flipping:

$$b := P(s), Y(s_p) \leftrightarrow P(s^1), Y(s_p^1)$$
$$f := P(f_0) \leftrightarrow P(f_1)$$

Note that there is a design choice here. In effect, we are saying that we are not interested in forming/breaking the bonds between the Ps in the ring. If we

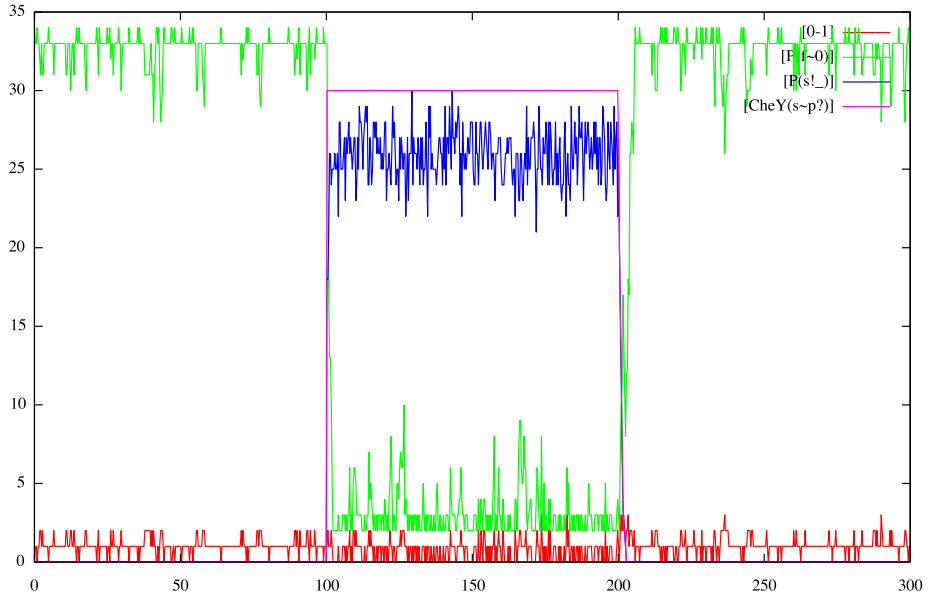


Fig. 1. The simulation steps up the amount of active Y at $t = 100$, and down again at $t = 200$; this sends the entire ring into an homogeneously 1 conformation, and back to 0. The number of mismatches (lowest curve) stays low, even during transitions.

wanted to incorporate also the ring assembly in the model, we would have to add $P(x), P(y) \leftrightarrow P(x^1), P(y^1)$ among our generator set $\mathcal{G}$. This would generate many more refined rules, as we will see. Recall that our patterns fall in three subgroups: $P(f_i, x^1), P(y^1, f_j)$; $P(f_i)$; and $P(f_i, s^1), Y(s^1)$.

Consider the extensions of b: clearly only the last pattern can glue relevantly on it; the corresponding (unique) site request is for P to reveal f and its internal state. This gives the first two rules b_0, b_1.

Consider now the more interesting extensions of f: the second pattern type glues relevantly but does not generate any site request; the third one asks P to reveal its site s, resulting in two possible extensions (s- means that s is bound):

$$P(f_0, s) \leftrightarrow P(f_1, s)$$
$$P(f_0, s\text{-}) \leftrightarrow P(f_1, s\text{-})$$

These extensions are *not* mature yet, as one can glue relevantly patterns of the first type on both sides of P, inducing a further request for revealing P's sites x and y. If we are in the component of an initial state where Ps are arranged in a ring, then we know that the neighbours on both sides exist and are Ps; this gives the final refinement of the above into the rules f_{ij}, f'_{ij} described earlier. If, on the other hand, we do not know that, we also have to add several rules where one or both of x, y are free, corresponding to open P-chains. This demonstrates the sensitivity of the obtained rule set to the initial choice of generators.

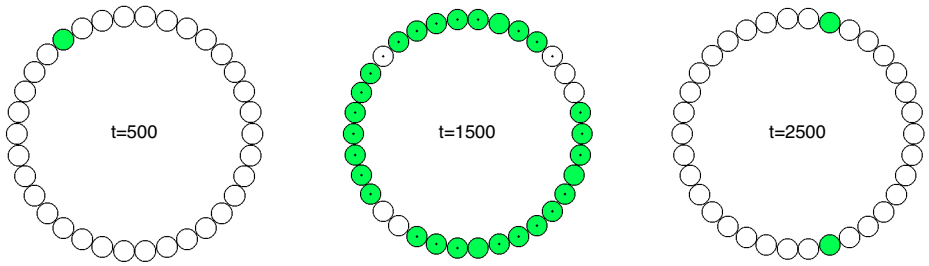


Fig. 2. Snapshots of the ring configuration are taken at time 500, 1500, and 2500. Solid (green) circles indicate conformation 1, hollow ones conformation 0; a dot in the centre indicates a bound (hence active) Y. At times 500, 2500, no Y is bound (because they are all inactive) and the ring is globally in state 0, up to tiny fluctuations; at time 1500, it is globally in state 1 as a consequence of the binding of Ys.

Hence, the rule set above does accord with our general refinement strategy of §3.

We can visualize the obtained simulations by extracting snapshots before, after and during the injection of active Ys, as in Fig. 2. Again we see few mismatches in both regime because of the Ising interaction expressed by the ϵ^{PP} energy costs. The full model is available on-line at `http://www.rulebase.org/models/ising-ring`. The choice of rates made in Ref. [2] for the f-generator is a particular symmetric case of our model (7), namely $A_{f^\star} = A_f = I/2$.

5 Conclusion

We have presented a new 'energy-oriented' methodology for the development of site graph rewriting models based on a set $\mathcal{P}$ of energy patterns; these patterns use a graphical syntax which allows us to specify the energy landscape. Rewrite rules are implicitly defined by $\mathcal{P}$ and generator rules $\mathcal{G}$. The resulting rule set $\mathcal{G}_\mathcal{P}$ is guaranteed to be thermodynamically correct and to eventually converge to the probability distribution described by the energy landscape given suitable rates. The construction is entirely parametric in the energy costs ϵ, and modular in $\mathcal{G}$. This means that in a modelling context, one can sweep over various values for ϵ without having to rebuild the model, and compositionally add new rule components to $\mathcal{G}$. Both features are clearly useful.

We expect our construction to provide a broad and uniform language to describe and analyse models of interacting biomolecules and similar systems of a quantitative fine-grained and distributed nature.

There are no specific conditions bearing on this construction other than that energy patterns should be local. It would be interesting to investigate whether, suitable constraints on patterns and generator rules can obtain optimized generated rule sets. Another interesting extension would be to deal with non-local

forms of energies expressing long-range interactions, where the metric is read off the graph itself. In practice, there will be many more rules generated, and beyond the descriptive aspects, simulations will need new ideas to be feasible. A ray of hope comes from the log-affine kinetic model (presented in the last subsection), as rules can be partitioned by energy balances for faster selection.

Finally, as said in the introduction, there is a growing body of literature which turns a theoretical eye to site graph rewriting [14,10,15,5], and it is tempting to ask whether our derivation can be replayed in more abstract settings; in particular, it would be very interesting to investigate its integration with the abstract framework for rule-based modelling developed in [21].

Acknowledgments. We would like to thank Peter Swain, Andrea Weisse, Julien Ollivier, Nicolas Oury, Finlo Boyde and Eric Deeds for many useful and fascinating conversations concerning the subject of this paper.

References

1. Bachman, J.A., Sorger, P.: New approaches to modeling complex biochemistry. Nature Methods 8(2), 130 (2011)
2. Bai, F., Branch, R.W., Nicolau Jr., D.V., Pilizota, T., Steel, B.C., Maini, P.K., Berry, R.M.: Conformational spread as a mechanism for cooperativity in the bacterial flagellar switch. Science 327(5966), 685–689 (2010)
3. Bournez, O., Côme, G.-M., Conraud, V., Kirchner, H., Ibanescu, L.: A rule-based approach for automated generation of kinetic chemical mechanisms. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 30–45. Springer, Heidelberg (2003)
4. Danos, V., Feret, J., Fontana, W., Krivine, J.: Scalable simulation of cellular signaling networks. In: Asian Symposium on Programming Languages and Systems, pp. 139–157 (2007)
5. Danos, V., Harmer, R., Winskel, G.: Constraining rule-based dynamics with types. Mathematical Structures in Computer Science 23(2), 272–289 (2013)
6. Danos, V., Oury, N.: Equilibrium and termination II: the case of Petri Nets. Mathematical Structures in Computer Science 23(2), 290–307 (2013)
7. Danos, V.: Agile modelling of cellular signalling. SOS 2008 Invited paper, Electronic Notes in Theoretical Computer Science 229(4), 3–10 (2009)
8. Danos, V., Oury, N.: Equilibrium and termination. In: Barry Cooper, S., Panangaden, P., Kashefi, E. (eds.) Proceedings Sixth Workshop on Developments in Computational Models: Causality, Computation, and Physics. EPTCS, vol. 26, pp. 75–84 (2010)
9. Diers, Y.: Familles universelles de morphismes. Tech. report, Université des Sciences et Techniques de Lille I (1978)
10. Dixon, L., Kissinger, A.: Open graphs and monoidal theories. arXiv:1011.4114 (2010)
11. Ehrig, H.: Handbook of graph grammars and computing by graph transformation: Applications, Languages and Tools, vol. 2. World Scientific Publishing Company (1999)
12. Faeder, J.R., Blinov, M.L., Hlavacek, W.S.: Rule-based modeling of biochemical systems with BioNetGen. Methods Mol. Biol. 500, 113–167 (2009)

13. Gross, T., Sayama, H.: Adaptive networks. Springer (2009)
14. Hayman, J., Heindel, T.: Pattern graphs and rule-based models: The semantics of kappa. In: Pfenning, F. (ed.) FOSSACS 2013. LNCS, vol. 7794, pp. 1–16. Springer, Heidelberg (2013)
15. Heckel, R.: DPO transformation with open maps. Graph Transformations, 203–217 (2012)
16. Heckel, R.: Dpo transformation with open maps. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 203–217. Springer, Heidelberg (2012)
17. Hlavacek, W.S., Faeder, J.R., Blinov, M.L., Posner, R.G., Hucka, M., Fontana, W.: Rules for modeling signal-transduction systems. Science Signalling 2006(344) (2006)
18. Krivine, J., Milner, R., Troina, A.: Stochastic bigraphs. Electronic Notes in Theoretical Computer Science 218, 73–96 (2008)
19. Lack, S., Sobociński, P.: Adhesive categories. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 273–288. Springer, Heidelberg (2004)
20. Lopez, C.F., Muhlich, J.L., Bachman, J.A., Sorger, P.K.: Programming biological models in python using pysb. Molecular Systems Biology 9(1) (2013)
21. Lynch, J.: A logical characterization of individual-based models. In: Proceedings of Logic in Computer Science, pp. 203–217 (2008)
22. Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., Teller, E., et al.: Equation of state calculations by fast computing machines. The Journal of Chemical Physics 21(6), 1087 (1953)
23. Murphy, E., Danos, V., Feret, J., Harmer, R., Krivine, J.: Rule-based modelling and model resolution. In: Lohdi, H., Muggleton, S. (eds.) Elements of Computational Systems Biology. Wiley (2010)
24. Tiger, C.-F., Krause, F., Cedersund, G., Palmér, R., Klipp, E., Hohmann, S., Kitano, H., Krantz, M.: A framework for mapping, visualisation and automatic model creation of signal-transduction networks. Molecular Systems Biology 8(1) (2012)

Globally Governed Session Semantics

Dimitrios Kouzapas and Nobuko Yoshida

Imperial College London

Abstract. This paper proposes a new bisimulation theory based on multiparty session types where a choreography specification governs the behaviour of session typed processes and their observer. The bisimulation is defined with the observer cooperating with the observed process in order to form complete global session scenarios and usable for proving correctness of optimisations for globally coordinating threads and processes. The induced bisimulation is strictly more fine-grained than the standard session bisimulation. The difference between the governed and standard bisimulations only appears when more than two interleaved multiparty sessions exist. The compositionality of the governed bisimilarity is proved through the soundness and completeness with respect to the governed reduction-based congruence.

1 Introduction

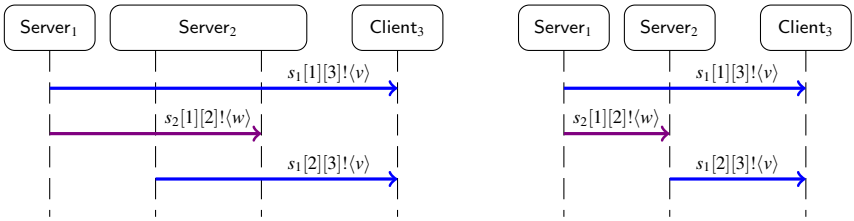
Modern society increasingly depends on distributed software infrastructures such as the backend of popular Web portals, global E-science cyberinfrastructure, e-healthcare and e-governments. An application in these environments is typically organised into many components which communicate through message passing. Thus an application is naturally designed as a collection of interaction scenarios, or *multiparty sessions*, each following an interaction pattern, or *choreographic protocol*. The theories of multiparty session types [11] capture these two natural abstraction units, representing the situation where two or more multiparty sessions (choreographies) can interleave for a single point application, with each message clearly identifiable as belonging to a specific session.

This paper introduces a new behavioural theory which can reason about distributed processes globally controlled by multiple choreographic sessions. Typed behavioural theory has been one of the central topics of the study of the π-calculus throughout its history, for example, in order to reason about various encodings into the typed π-calculi [16, 18]. Our theory treats the mutual effects of multiple choreographic sessions which are shared among distributed participants as their common knowledges or agreements, reflecting the origin of choreographic frameworks [5]. These features make our theory distinct from any type-based bisimulations in the literature and the theory applicable to real choreographic usecase from a large-scale distributed system. Since our bisimulation is based on the regulation of conversational behaviours of distributed components by global specifications, we call our bisimulation *globally governed bisimulation*.

To illustrate the key idea, we first explain the mechanisms of multiparty session types [11]. Let us consider a simple protocol where participant 1 sends a message of type `bool` to participant 2. To develop the code for this protocol, we start by specifying the *global type* [11] as $G_1 = 1 \to 2 : \langle \texttt{bool} \rangle . \texttt{end}$ where $\to$ signifies the flow of communication and `end` denotes protocol termination. With agreement on G_1 as a specification

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 395–409, 2013.

for participant 1 and participant 2, each program can be implemented separately. Then for type-checking, G_1 is *projected* into local session types: one local session type from 1's point of view, $[2]!\langle \texttt{bool} \rangle$ (output to 2 with `bool`-type), and another from 2's point of view, $[1]?\langle \texttt{bool} \rangle$ (input from 1 with `bool`-type), against which both processes are checked to be correct.



Resource Managment Example: (a) before optimisation; (b) after optimisation

Now we explain how our new theory can reason about an optimisation of choreography interactions (a simplified usecase (UC.R2.13 "Acquire Data From Instrument") from [1]). Consider the two global types between three participants $(1,2,3)$:

$$G_a = 1 \to 3 : \langle ser \rangle.2 \to 3 : \langle ser \rangle.\texttt{end}, \quad G_b = 1 \to 2 : \langle sig \rangle.\texttt{end}$$

and a scenario in the diagram (a) where Client3 (participant 3) uses two services, the first from Server1 (participant 1) and Server2 (participant 2), and Server1 sends an internal signal to Server2. The three parties belonging to these protocols are implemented as:

$$P_1 = a[1](x).b[1](y).x[3]!\langle v \rangle; y[2]!\langle w \rangle; \mathbf{0} \quad P_2 = a[2](x).\overline{b}[2](y).(y[1]?(z); \mathbf{0} \mid x[3]!\langle v \rangle; \mathbf{0})$$
$$P_3 = \overline{a}[3](x).x[1]?(z); x[2]?(y); \mathbf{0}$$

where session name a establishes the session corresponding to G_a. Client3 (P_3) initiates a session involving three processes as the third participant by $\overline{a}[3](x)$: Service1 (P_1) and Service2 (P_2) participate to the session $a[1](x)$ and $a[2](x)$, respectively. Similarly the session corresponding to G_b is established between Service1 and Service2.

Since from Client3, the internal signal is invisible, we optimise Server2 to a single thread to avoid an unnecessary thread creation as $R_2 = a[2](x).\overline{b}[2](y).y[1]?(z); x[3]!\langle v \rangle; \mathbf{0}$ in in the diagram (b). Note that both P_2 and R_2 are typable under G_a and G_b. Obviously, in the untyped setting, $P_1 \mid P_2$ and $P_1 \mid R_2$ are *not* bisimilar since in P_2, the output action $x[3]!\langle v \rangle$ can be observed before the input action $y[1]?(z)$ happens. However, with the global constraints given by G_a and G_b, a service provided by Server2 is only available to Client3 after Server1 sends a signal to Server2, i.e. action $x[3]!\langle v \rangle$ can only happen after action $y[1]?(z)$ in P_2. Hence $P_1 \mid P_2$ and $P_1 \mid R_2$ are not distinguishable by Client3 and the thread optimisation of R_2 is correct.

On the other hand, if we change the global type G_a as:

$$G'_a = 2 \to 3 : \langle ser \rangle.1 \to 3 : \langle ser \rangle.\texttt{end}$$

then R_2 can perform both the output to Client3 and the input from Server1 concurrently since G'_a states that Client3 can receive the message from Server2 first. Hence $P_1 \mid P_2$ and $P_1 \mid R_2$ are no longer equivalent.

The key point to make this difference possible is to observe the behaviour of processes together with the information provided by the global types. The global types can define additional knowledge about how the observer (the client in the above example) will collaborate with the observed processes so that different global types (i.e. global witnesses) can induce the different equivalences.

Contributions. This paper introduces two kinds of typed bisimulations based on multiparty session types. The first bisimulation is solely based on local (endpoint) types defined without global information, hence it resembles the standard linearity-based bisimulation. The second one is a *globally governed session bisimilarity* which uses multiparty session types as information for a global witness. We prove that each coincides with a corresponding standard contextual equivalence [10] (Theorems 3.1 and 4.1). The governed bisimulation gives more fine-grained equivalences than the locally typed bisimulation. We identify the condition when the two semantics exactly coincide (Theorem 4.2). Interestingly our theorem (Theorem 4.3) shows this difference appears only when processes are running under more than two interleaved global types. This feature makes the theory applicable to real situations where multiple choreographies are used in a single, large application. We demonstrate the use of governed bisimulation through the running example, which is applicable to a thread optimisation of a real usecase from a large scale distributed system [1]. The appendix includes auxiliary definitions, the full proofs and a full derivation of a usecase from [1].

2 Synchronous Multiparty Sessions

This section defines a synchronous version of the multiparty session types. The syntax and typing follows [4] except we eliminate queues for asynchronous communication. We chose synchrony since it allows the simplest formulations for demonstrating the essential concepts of bisimulations. The extension to asynchrony is given in [7].

Syntax. Below we define the syntax of the synchronous multiparty session calculus.

P	$::= \overline{u}[\mathtt{p}](x).P$	Request	$\mid$ $\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$	Conditional
	$\mid$ $u[\mathtt{p}](x).P$	Accept	$\mid$ $P \mid Q$	Parallel
	$\mid$ $c[\mathtt{p}]!\langle e\rangle;P$	Sending	$\mid$ $\mathbf{0}$	Inaction
	$\mid$ $c[\mathtt{p}]?(x);P$	Receiving	$\mid$ $(\nu\ n)P$	Hiding
	$\mid$ $c[\mathtt{p}] \oplus l;P$	Selection	$\mid$ $\mu X.P$	Recursion
	$\mid$ $c[\mathtt{p}]\&\{l_i : P_i\}_{i\in I}$	Branching	$\mid$ X	Variable
u	$::= x \mid a$	Identifier	$c ::= s[\mathtt{p}] \mid x$	Session
n	$::= s \mid a$	Name	$v ::= a \mid \mathtt{tt} \mid \mathtt{ff} \mid s[\mathtt{p}]$	Value
e	$::= v \mid x \mid e\ \mathtt{and}\ e' \mid e = e' \mid \ldots$	Expression		

Note that expressions includes name matching ($n = n$). We call $\mathtt{p}, \mathtt{p}', \mathtt{q}, \ldots$ (ranging over the natural numbers) the *participants*. For the primitives for session initiation, $\overline{u}[\mathtt{p}](x).P$ initiates a new session through an identifier u (which represents a shared interaction point) with the other multiple participants, each of shape $u[\mathtt{p}](x).Q_{\mathtt{q}}$ where $1 \leq \mathtt{q} \leq \mathtt{p} - 1$. The (bound) variable x is the channel used to do the communications.

Session communications (communications that take place inside an established session) are performed using the next two pairs: the sending and receiving of a value and the selection and branching (where the former chooses one of the branches offered by the latter). Process $c[\mathtt{p}]!\langle e\rangle;P$ sends a value to $\mathtt{p}$; accordingly, process $c[\mathtt{p}]?(x);P$ denotes the intention of receiving a value from the participant $\mathtt{p}$. The same holds for selection/branching. Process $\mathbf{0}$ is the inactive process. Other processes are standard. We say that a process is closed if it does not contain free variables. $\mathtt{fn}(P)/\mathtt{bn}(P)$ and $\mathtt{fv}(P)/\mathtt{bv}(P)$ denote a set of free/bound names and free/bound variables, respectively. We use the standard structure rules (denoted by $\equiv$) including $\mu X.P \equiv P\{\mu X.P/X\}$.

Operational Semantics. Operational semantics of the calculus are defined below.

$$a[1](x).P_1 \mid \Pi_{i=\{2,..,n\}} a[i](x).P_i \longrightarrow (\nu\, s)(P_1\{s[1]/x\} \mid \Pi_{i=\{2,..,n\}} P_i\{s[i]/x\}) \quad \text{[Link]}$$

$$s[\mathtt{p}][\mathtt{q}]!\langle e\rangle;P \mid s[\mathtt{q}][\mathtt{p}]?(x);Q \longrightarrow P \mid Q\{v/x\} \quad (e \downarrow v) \quad \text{[Comm]}$$

$$s[\mathtt{p}][\mathtt{q}] \oplus l_k;P \mid s[\mathtt{q}][\mathtt{p}]\&\{l_i : P_i\}_{i\in I} \longrightarrow P \mid P_k \quad (k \in I) \quad \text{[Label]}$$

$$\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow P \quad (e \downarrow \mathtt{tt}) \qquad \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow Q \quad (e \downarrow \mathtt{ff}) \quad \text{[If]}$$

$$\frac{P \longrightarrow P'}{(\nu\, n)P \longrightarrow (\nu\, n)P'}\ \text{[Res]} \qquad \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}\ \text{[Par]} \qquad \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}\ \text{[Str]}$$

Rule [Link] defines synchronous session initiation. All session roles must be present to synchronously reduce each role $\mathtt{p}$ on a fresh session name $s[\mathtt{p}]$. Rule [Comm] is for sending a value to the corresponding receiving process where $e \downarrow v$ means expression e evaluates to value v. The interaction between selection and branching is defined via rule [Label]. Other rules are standard. We write $\twoheadrightarrow$ for $(\longrightarrow \cup \equiv)^*$.

Global types, ranged over by $G, G', \ldots$ describe the whole conversation scenario of a multiparty session as a type signature. Its grammar is given below.

Global	G	$::=$	$\mathtt{p} \to \mathtt{q} : \langle U\rangle.G'$	exchange	Local T	$::=$	$[\mathtt{p}]!\langle U\rangle;T$ send
		$\mid$	$\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i\in I}$	branching		$\mid$	$[\mathtt{p}]?(U);T$ receive
		$\mid$	$\mu \mathtt{t}.G$	recursion		$\mid$	$[\mathtt{p}] \oplus \{l_i : T_i\}_{i\in I}$ selection
		$\mid$	$\mathtt{t}$	variable		$\mid$	$[\mathtt{p}]\&\{l_i : T_i\}_{i\in I}$ branching
		$\mid$	$\mathtt{end}$	end		$\mid$	$\mu \mathtt{t}.T$ recursion
Exchange	U	$::=$	$S \mid T$			$\mid$	$\mathtt{t}$ variable
Sort	S	$::=$	$\mathtt{bool} \mid \langle G\rangle$			$\mid$	$\mathtt{end}$ end

The global type $\mathtt{p} \to \mathtt{q} : \langle U\rangle.G'$ says that participant $\mathtt{p}$ sends a message of type U to the participant $\mathtt{q}$ and then interactions described in G' take place. *Exchange types* $U, U', \ldots$ consist of *sorts* types $S, S', \ldots$ for values (either base types or global types), and *local session* types $T, T', \ldots$ for channels (defined in the next paragraph). Type $\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i\in I}$ says participant $\mathtt{p}$ sends one of the labels l_i to $\mathtt{q}$. If l_j is sent, interactions described in G_j take place. In both cases we assume $\mathtt{p} \neq \mathtt{q}$. Type $\mu \mathtt{t}.G$ is a recursive type, assuming type variables $(\mathtt{t}, \mathtt{t}', \ldots)$ are guarded in the standard way, i.e., type variables only appear under some prefix. We take an *equi-recursive* view of recursive types, not distinguishing between $\mu \mathtt{t}.G$ and its unfolding $G\{\mu \mathtt{t}.G/\mathtt{t}\}$.We assume that G in the grammar of sorts has no free type variables. Type $\mathtt{end}$ represents the termination of the session.

Local types correspond to the communication actions, representing sessions from the view-points of single participants. The *send type* $[\mathtt{p}]!\langle U\rangle;T$ expresses the sending to $\mathtt{p}$ of a value of type U, followed by the communications of T. The *selection type* $[\mathtt{p}]\oplus\{l_i:T_i\}_{i\in I}$ represents the transmission to $\mathtt{p}$ of a label l_i chosen in the set $\{l_i \mid i\in I\}$ followed by the communications described by T_i. The *receive* and *branching* are dual. Other types are the same as global types.

The relation between global and local types is formalised by the standard projection function [11]. For example, $(\mathtt{p}'\to\mathtt{q}:\langle U\rangle.G)\lceil\mathtt{p}$ is defined as: $[\mathtt{q}]!\langle U\rangle;(G\lceil\mathtt{p})$ if $\mathtt{p}=\mathtt{p}'$, $[\mathtt{p}']?(U);(G\lceil\mathtt{p})$ if $\mathtt{p}=\mathtt{q}$ and $G\lceil\mathtt{p}$ otherwise. Then the *projection set* of $s:G$ is defined as $\mathtt{proj}(s:G)=\{s[\mathtt{p}]:G\lceil\mathtt{p} \mid \mathtt{p}\in\mathtt{roles}(G)\}$ where $\mathtt{roles}(G)$ denotes the set of the roles appearing in G.

Typing System. The typing judgements for expressions and processes are of the shapes:

$$\Gamma\vdash e:S \quad \text{and} \quad \Gamma\vdash P\triangleright\Delta$$

where Γ is the standard environment which associates variables to sort types, shared names to global types and process variables to session environments; and Δ is the session environment which associates channels to session types. Formally we define: $\Gamma ::= \emptyset \mid \Gamma\cdot u:S \mid \Gamma\cdot X:\Delta$ and $\Delta ::= \emptyset \mid \Delta\cdot c:T$, assuming we can write $\Gamma\cdot u:S$ if $u\notin\mathtt{dom}(\Gamma)$. We extend this to a concatenation for typing environments as $\Delta\cdot\Delta'=\Delta\cup\Delta'$. Typing Δ is *coherent with respect to session* s (notation $\mathtt{co}(\Delta(s))$) if for all $s[\mathtt{p}]:T_\mathtt{p}, s[\mathtt{q}]:T_\mathtt{q}\in\Delta$, $T_\mathtt{p}$ and $T_\mathtt{q}$ are dual each other (it is given by exchanging ! by ? and $\oplus$ by & [9]). A typing Δ is *coherent* (notation $\mathtt{co}(\Delta)$) if it is coherent with respect to all s in its domain. The typing judgement $\Gamma\vdash P\triangleright\Delta$ is *coherent* if $\mathtt{co}(\Delta)$.

The typing rules are essentially identical to the communication typing system for programs in [4] (since we do not require queues). The rest of the paper can be read without knowing the typing system.

Type Soundness. Next we define the reduction semantics for local types. Since session environments represent the forthcoming communications, by reducing processes session environments can change. This can be formalised as in [4, 11] by introducing the notion of reduction of session environments, whose rules are:

1. $\{s[\mathtt{p}]:[\mathtt{q}]!\langle U\rangle;T\cdot s[\mathtt{q}]:[\mathtt{p}]?(U);T'\}\longrightarrow\{s[\mathtt{p}]:T\cdot s[\mathtt{q}]:T'\}$.
2. $\{s[\mathtt{p}]:[\mathtt{q}]\oplus\{l_i:T_i\}_{i\in I}\cdot s[\mathtt{q}]:[\mathtt{p}]\&\{l_j:T'_j\}_{j\in J}\}\longrightarrow\{s[\mathtt{p}]:T_k\cdot s[\mathtt{q}]:T'_k\}$ $I\subseteq J, k\in I$.
3. $\Delta\cup\Delta'\longrightarrow\Delta\cup\Delta''$ if $\Delta'\longrightarrow\Delta''$.

We write $\twoheadrightarrow=\longrightarrow^*$. Note that $\Delta\twoheadrightarrow\Delta'$ is non-deterministic (i.e. not always confluent) by the second rule. Then the typing system satisfies the subject reduction theorem [4]: if $\Gamma\vdash P\triangleright\Delta$ is coherent and $P\twoheadrightarrow P'$ then $\Gamma\vdash P'\triangleright\Delta'$ is coherent with $\Delta\twoheadrightarrow\Delta'$.

3 Synchronous Multiparty Session Semantics

This section presents the standard typed behavioural theory for the synchronous multiparty sessions.

Labels. We use the following labels $(\ell,\ell',\ldots)$:

$$\begin{array}{rcl}\ell & ::= & \overline{a}[A](s) \mid a[A](s) \mid s[\mathtt{p}][\mathtt{q}]!\langle v\rangle \mid s[\mathtt{p}][\mathtt{q}]!(a)\\ & \mid & s[\mathtt{p}][\mathtt{q}]!(s'[\mathtt{p}']) \mid s[\mathtt{p}][\mathtt{q}]?\langle v\rangle \mid s[\mathtt{p}][\mathtt{q}]\oplus l \mid s[\mathtt{p}][\mathtt{q}]\&l \mid \tau\end{array}$$

$$\langle\text{Req}\rangle \quad \overline{a}[\mathtt{p}](x).P \xrightarrow{\overline{a}[\{\mathtt{p}\}](s)} P\{s[\mathtt{p}]/x\} \qquad \langle\text{Acc}\rangle \quad a[\mathtt{p}](x).P \xrightarrow{a[\{\mathtt{p}\}](s)} P\{s[\mathtt{p}]/x\}$$

$$\langle\text{Send}\rangle \quad s[\mathtt{p}][\mathtt{q}]!\langle e\rangle;P \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle v\rangle} P \quad (e\downarrow v) \qquad \langle\text{Rcv}\rangle \quad s[\mathtt{p}][\mathtt{q}]?(x);P \xrightarrow{s[\mathtt{p}][\mathtt{q}]?\langle v\rangle} P\{v/x\}$$

$$\langle\text{Sel}\rangle \quad s[\mathtt{p}][\mathtt{q}]\oplus l;P \xrightarrow{s[\mathtt{p}][\mathtt{q}]\oplus l} P \qquad \langle\text{Bra}\rangle \quad s[\mathtt{p}][\mathtt{q}]\&\{l_i:P_i\}_{i\in I} \xrightarrow{s[\mathtt{p}][\mathtt{q}]\&l_k} P_k$$

$$\langle\text{Tau}\rangle \frac{P \xrightarrow{\ell} P' \quad Q \xrightarrow{\ell'} Q' \quad \ell \asymp \ell'}{P \mid Q \xrightarrow{\tau} (\nu\ \mathtt{bn}(\ell)\cap\mathtt{bn}(\ell'))(P' \mid Q')} \qquad \langle\text{Par}\rangle \frac{P \xrightarrow{\ell} P' \quad \mathtt{bn}(\ell)\cap\mathtt{fn}(Q)=\emptyset}{P \mid Q \xrightarrow{\ell} P' \mid Q}$$

$$\langle\text{Res}\rangle \frac{P \xrightarrow{\ell} P' \quad n \notin \mathtt{fn}(\ell)}{(\nu\ n)P \xrightarrow{\ell} (\nu\ n)P'} \quad \langle\text{OpenS}\rangle \frac{P \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle s'[\mathtt{p}']\rangle} P'}{(\nu\ s')P \xrightarrow{s[\mathtt{p}][\mathtt{q}]!(s'[\mathtt{p}'])} P'} \quad \langle\text{OpenN}\rangle \frac{P \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle a\rangle} P'}{(\nu\ a)P \xrightarrow{s[\mathtt{p}][\mathtt{q}]!(a)} P'}$$

$$\langle\text{Alpha}\rangle \frac{P \equiv_\alpha P' \quad P' \xrightarrow{\ell} Q'}{P \xrightarrow{\ell} Q} \qquad \langle\text{AcPar}\rangle \frac{P_1 \xrightarrow{a[A](s)} P_1' \quad P_2 \xrightarrow{a[A'](s)} P_2' \quad A\cap A'=\emptyset}{P_1 \mid P_2 \xrightarrow{a[A\cup A'](s)} P_1' \mid P_2'}$$

$$\langle\text{ReqPar}\rangle \frac{P_1 \xrightarrow{a[A](s)} P_1' \quad P_2 \xrightarrow{\overline{a}[A'](s)} P_2' \quad A\cap A'=\emptyset,\ A\cup A' \text{ not complete w.r.t } \max(A')}{P_1 \mid P_2 \xrightarrow{\overline{a}[A\cup A'](s)} P_1' \mid P_2'}$$

$$\langle\text{TauS}\rangle \frac{P_1 \xrightarrow{a[A](s)} P_1' \quad P_2 \xrightarrow{\overline{a}[A'](s)} P_2' \quad A\cap A'=\emptyset,\ A\cup A' \text{ complete w.r.t } \max(A')}{P_1 \mid P_2 \xrightarrow{\tau} (\nu\ s)(P_1' \mid P_2')}$$

We omit the synmetric case of ⟨Par⟩ and conditionals.

Fig. 1. Labelled transition system for processes

A role set A is a set of multiparty session types roles. Labels $\overline{a}[A](s)$ and $a[A](s)$ define the accept and request of a fresh session s by roles in set A respectively. Actions on session channels are denoted with labels $s[\mathtt{p}][\mathtt{q}]!\langle v\rangle$ and $s[\mathtt{p}][\mathtt{q}]?\langle v\rangle$ for output and input of value v from $\mathtt{p}$ to $\mathtt{q}$ on session s. Bound output values can be shared channels or session roles (delegation). $s[\mathtt{p}][\mathtt{q}]\oplus l$ and $s[\mathtt{p}][\mathtt{q}]\&l$ define the selection and branching respectively. Label τ is the standard hidden transition.

Dual label definition is used to define the parallel rule in the label transition system:

$$s[\mathtt{p}][\mathtt{q}]!\langle v\rangle \asymp s[\mathtt{q}][\mathtt{p}]?\langle v\rangle \quad s[\mathtt{p}][\mathtt{q}]!(v) \asymp s[\mathtt{q}][\mathtt{p}]?\langle v\rangle \quad s[\mathtt{p}][\mathtt{q}]\oplus l \asymp s[\mathtt{q}][\mathtt{p}]\&l$$

Dual labels are input and output (resp. selection and branching) on the same session channel and on complementary roles. For example, in $s[\mathtt{p}][\mathtt{q}]!\langle v\rangle$ and $s[\mathtt{q}][\mathtt{p}]?\langle v\rangle$, role $\mathtt{p}$ sends to $\mathtt{q}$ and role $\mathtt{q}$ receives from $\mathtt{p}$. Another important definition for the session initiation is the notion of the complete role set. We say the role set A is *complete with respect to* n if $n=\max(A)$ and $A=\{1,2,\ldots,n\}$. The complete role set means that all global protocol participants are present in the set. For example, $\{1,3,4\}$ is not complete, but $\{1,2,3,4\}$ is. We use $\mathtt{fn}(\ell)$ and $\mathtt{bn}(\ell)$ to denote a set of free and bound names in ℓ and set $\mathtt{n}(\ell)=\mathtt{bn}(\ell)\cup\mathtt{fn}(\ell)$.

Labelled Transition System for Processes. Figure 1 gives the untyped labelled transition system. Rules ⟨Req⟩ and ⟨Acc⟩ define the accept and request actions for a fresh session s on role $\{\mathtt{p}\}$. Rules ⟨Send⟩ and ⟨Rcv⟩ give the send and receive respectively for value v from role $\mathtt{p}$ to role $\mathtt{q}$ in session s. Rules ⟨Sel⟩ and ⟨Bra⟩ define selecting and branching labels.

$$
\begin{array}{rrcl}
\Gamma(a) = \langle G \rangle, s \text{ fresh implies} & (\Gamma, \Delta) & \xrightarrow{a[A](s)} & (\Gamma, \Delta \cdot \{s[i] : G\lceil i\}_{i \in A}) \\
\Gamma(a) = \langle G \rangle, s \text{ fresh implies} & (\Gamma, \Delta) & \xrightarrow{\overline{a}[A](s)} & (\Gamma, \Delta \cdot \{s[i] : G\lceil i\}_{i \in A}) \\
\Gamma \vdash v : U, s[\mathtt{q}] \notin \mathtt{dom}(\Delta) \text{ implies} & (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]!\langle U \rangle; T) & \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle v \rangle} & (\Gamma, \Delta \cdot s[\mathtt{p}] : T) \\
s[\mathtt{q}] \notin \mathtt{dom}(\Delta), a \notin \mathtt{dom}(\Gamma) \text{ implies} & (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]!\langle U \rangle; T) & \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle a \rangle} & (\Gamma \cdot a : U, \Delta \cdot s[\mathtt{p}] : T) \\
\Gamma \vdash v : U, s[\mathtt{q}] \notin \mathtt{dom}(\Delta) \text{ implies} & (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]?(U); T) & \xrightarrow{s[\mathtt{p}][\mathtt{q}]?\langle v \rangle} & (\Gamma, \Delta \cdot s[\mathtt{p}] : T) \\
a \notin \mathtt{dom}(\Gamma), s[\mathtt{q}] \notin \mathtt{dom}(\Delta) \text{ implies} & (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]?(U); T) & \xrightarrow{s[\mathtt{p}][\mathtt{q}]?\langle a \rangle} & (\Gamma \cdot a : U, \Delta \cdot s[\mathtt{p}] : T) \\
s[\mathtt{q}] \notin \mathtt{dom}(\Delta) \text{ implies} & (\Gamma, \Delta \cdot s'[\mathtt{p}'] : T' \cdot s[\mathtt{p}] : [\mathtt{q}]!\langle T' \rangle; T) & \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle s'[\mathtt{p}'] \rangle} & (\Gamma, \Delta \cdot s[\mathtt{p}] : T) \\
s[\mathtt{q}] \notin \mathtt{dom}(\Delta) \text{ implies} & (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]!\langle T' \rangle; T) & \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle s'[\mathtt{p}'] \rangle} & (\Gamma, \Delta \cdot s[\mathtt{p}] : T \cdot \{s'[\mathtt{p}_i] : T_i\}) \\
s[\mathtt{q}], s'[\mathtt{p}'] \notin \mathtt{dom}(\Delta) \text{ implies} & (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]?(T'); T) & \xrightarrow{s[\mathtt{p}][\mathtt{q}]?\langle s'[\mathtt{p}'] \rangle} & (\Gamma, \Delta \cdot s'[\mathtt{p}'] : T' \cdot s[\mathtt{p}] : T) \\
s[\mathtt{q}] \notin \mathtt{dom}(\Delta) \text{ implies} & (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}] \oplus \{l_i : T_i\}_{i \in I}) & \xrightarrow{s[\mathtt{p}][\mathtt{q}] \oplus l_k} & (\Gamma, \Delta \cdot s[\mathtt{p}] : T_k) \\
s[\mathtt{q}] \notin \mathtt{dom}(\Delta) \text{ implies} & (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}] \& \{l_i : T_i\}_{i \in I}) & \xrightarrow{s[\mathtt{p}][\mathtt{q}] \& l_k} & (\Gamma, \Delta \cdot s[\mathtt{p}] : T_k) \\
\Delta \longrightarrow \Delta' \quad \text{or} \quad \Delta = \Delta' \text{ implies} & (\Gamma, \Delta) & \xrightarrow{\tau} & (\Gamma, \Delta')
\end{array}
$$

Fig. 2. Labelled Transition Relation for Environments

The last three rules are for collecting and synchronising the multiparty participants together. Rule $\langle$AccPar$\rangle$ accumulates the accept participants and records them into role set A. Rule $\langle$ReqPar$\rangle$ accumulates the accept participants and the request participant into role set A. Note that the request action role set always includes the maximum role number among the participants. Finally, rule $\langle$TauS$\rangle$ checks that a role set is complete, thus a new session can be created under the τ-action (synchronisation). Other rules are standard. See Example 3.1. We write $\Longrightarrow$ for the reflexive and transitive closure of $\longrightarrow$, $\stackrel{\ell}{\Longrightarrow}$ for the transitions $\Longrightarrow\stackrel{\ell}{\longrightarrow}\Longrightarrow$ and $\stackrel{\hat{\ell}}{\Longrightarrow}$ for $\stackrel{\ell}{\Longrightarrow}$ if $\ell \neq \tau$ otherwise $\Longrightarrow$.

Typed Labelled Transition Relation. We define the typed LTS on the basis of the untyped one. This is realised by introducing the definition of an environment labelled transition system, defined in Figure 2. $(\Gamma, \Delta) \stackrel{\ell}{\longrightarrow} (\Gamma', \Delta')$ means that an environment (Γ, Δ) allows an action to take place, and the resulting environment is (Γ', Δ').

The intuition for this definition is that observables on session channels occur when the corresponding endpoint is not present in the linear typing environment Δ, and the type of an action's object respects the environment (Γ, Δ). In the case when new names are created or received, the environment (Γ, Δ) is extended.

The first rule says that reception of a message via a is possible when a's type $\langle G \rangle$ is recorded into Γ and the resulting session environment records projected types from G ($\{s[i] : G\lceil i\}_{i \in A}$). The second rule is for the send of a message via a and it is dual to the first rule. The next four rules are free value output, bound name output, free value input and name input. Rest of rules are free session output, bound session output, and session input as well as selection and branching rules. The bound session output records a set of session types $s'[\mathtt{p}_i]$ at opened session s'. The final rule ($\ell = \tau$) follows the reduction rules for linear session environment defined in § 2 ($\Delta = \Delta'$ is the case for the reduction at hidden sessions). Note that if Δ already contains destination ($s[\mathtt{q}]$), the environment cannot perform the visible action, but only the final τ-action.

The typed LTS requires that a process can perform an untyped action ℓ and that its typing environment (Γ, Δ) can match the action ℓ.

Definition 3.1 (Typed transition). *Typed transition* relation is defined as $\Gamma_1 \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} \Gamma_2 \vdash P_2 \triangleright \Delta_2$ if (1) $P_1 \xrightarrow{\ell} P_2$ and (2) $(\Gamma_1, \Delta_1) \xrightarrow{\ell} (\Gamma_2, \Delta_2)$ with $\Gamma_i \vdash P_i \triangleright \Delta_i$.

Synchronous Multiparty Behavioural Theory. We first define a relation $\mathscr{R}$ as *typed relation* if it relates two closed, coherent typed terms $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} \Gamma \vdash P_2 \triangleright \Delta_2$. We often write $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$.

Next we define the *barb* [2]: we write $\Gamma \vdash P \triangleright \Delta \downarrow_{s[\mathtt{p}][\mathtt{q}]}$ if $P \equiv (\nu\, \tilde{a}\tilde{s})(s[\mathtt{p}][\mathtt{q}]!\langle v\rangle; R \mid Q)$ with $s \notin \tilde{s}$ and $s[\mathtt{q}] \notin \mathtt{dom}(\Delta)$; and $\Gamma \vdash P \triangleright \Delta \downarrow_a$ if $P \equiv (\nu\, \tilde{a}\tilde{s})(\overline{a}[n](s).R \mid Q)$ with $a \notin \tilde{a}$. Then we write m for either a or $s[\mathtt{p}][\mathtt{q}]$. We define $\Gamma \vdash P \triangleright \Delta \Downarrow_m$ if there exists Q such that $P \twoheadrightarrow Q$ and $\Gamma \vdash Q \triangleright \Delta' \downarrow_m$.

We write $\Delta_1 \rightleftharpoons \Delta_2$ if there exists Δ such that $\Delta_1 \twoheadrightarrow \Delta$ and $\Delta_2 \twoheadrightarrow \Delta$. We now define the contextual congruence based on the barb and [10].

Definition 3.2 (Reduction congruence). A typed relation $\mathscr{R}$ is *reduction congruence* if it satisfies the following conditions for each $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$ with $\Delta_1 \rightleftharpoons \Delta_2$.

1. $\Gamma \vdash P_1 \triangleright \Delta_1 \Downarrow_m$ iff $\Gamma \vdash P_2 \triangleright \Delta_2 \Downarrow_m$
2. Whenever $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$ holds, $P_1 \twoheadrightarrow P_1'$ implies $P_2 \twoheadrightarrow P_2'$ such that $\Gamma \vdash P_1' \triangleright \Delta_1' \mathscr{R} P_2' \triangleright \Delta_2'$ holds with $\Delta_1' \rightleftharpoons \Delta_2'$.
3. For all closed context C, such that $\Gamma \vdash C[P_1'] \triangleright \Delta_1'$ and $\Gamma \vdash C[P_2'] \triangleright \Delta_2'$ where $\Delta_1' \rightleftharpoons \Delta_2'$, $\Gamma \vdash C[P_1] \triangleright \Delta_1' \mathscr{R} \Gamma \vdash C[P_2] \triangleright \Delta_2'$.

The union of all reduction congruence relations is denoted as $\cong^s$.

Definition 3.3 (Synchronous multiparty session bisimulation). A typed relation $\mathscr{R}$ over closed processes is a (weak) *synchronous multiparty session bisimulation* or often a *synchronous bisimulation* if, whenever $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$, it holds:

1. $\Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} \Gamma' \vdash P_1' \triangleright \Delta_1'$ implies $\Gamma \vdash P_2 \triangleright \Delta_2 \xRightarrow{\hat{\ell}} \Gamma' \vdash P_2' \triangleright \Delta_2'$ such that $\Gamma' \vdash P_1' \triangleright \Delta_1' \mathscr{R} P_2' \triangleright \Delta_2'$.
2. The symmetric case.

The maximum bisimulation exists which we call *synchronous bisimilarity*, denoted by $\approx^s$. We sometimes leave environments implicit, writing e.g. $P \approx^s Q$. We also write $\approx$ for untyped synchronous bisimilarity which is defined by the untyped LTS in Figure 1 but without the environment LTS in Figure 2.

Theorem 3.1 (Soundness and completeness). $\cong^s = \approx^s$.

Example 3.1 (Synchronous multiparty bisimulation). We use the running example from § 1. First we explain the session initialisation from Figure 1. By $\langle \mathrm{Acc} \rangle$ and $\langle \mathrm{Req} \rangle$,

$$P_1 \xrightarrow{a[\{1\}](s_1)} P_1' = b[1](y).s_1[1][3]!\langle v\rangle; y[2]!\langle w\rangle; \mathbf{0}$$
$$P_2 \xrightarrow{a[\{2\}](s_1)} P_2' = \overline{b}[2](y).(y[1]?(z); \mathbf{0} \mid s_1[2][3]!\langle v\rangle; \mathbf{0}) \quad P_3 \xrightarrow{\overline{a}[\{3\}](s_1)} P_3' = s_1[3][1]?(z); s_1[3][2]?(y); \mathbf{0}$$

with

$$\Gamma \vdash P_1' \triangleright s_1[1] : [3]!\langle U\rangle;\mathtt{end},\ \Gamma \vdash P_2' \triangleright s_1[2] : [3]!\langle U\rangle;\mathtt{end},\ \Gamma \vdash P_3' \triangleright s_1[3] : [1]?(U);[2]?(U);\mathtt{end}$$

By $\langle$AccPar$\rangle$, we have $P_1 \mid P_2 \xrightarrow{a[\{1,2\}](s_1)} P_1' \mid P_2'$. We have another possible initialisation: $P_1 \mid P_3 \xrightarrow{\overline{a}[\{1,3\}](s_1)} P_1' \mid P_3'$. From both of them, if we compose another process, the set $\{1,2,3\}$ becomes complete so that by synchronisation $\langle$TauS$\rangle$, $\Gamma \vdash P_1 \mid P_2 \mid P_3 \triangleright \emptyset \xrightarrow{\tau} \Gamma \vdash (\nu\, s_1)(P_1' \mid P_2' \mid P_3') \triangleright \emptyset$. Further we have:

$$\Gamma \vdash P_1' \mid P_2' \triangleright \Delta_0 \xrightarrow{\tau} \Gamma \vdash (\nu\, s_2)(s_1[1][3]!\langle v\rangle;s_2[1][2]!\langle w\rangle;\mathbf{0} \mid s_2[2][1]?(z);\mathbf{0} \mid s_1[2][3]!\langle v\rangle;\mathbf{0}) = Q_1 \triangleright \Delta_0$$

with $\Delta_0 = s_1[1] : [3]!\langle U\rangle;\mathtt{end} \cdot s_1[2] : [3]!\langle U\rangle;\mathtt{end}$. Then

$$\Gamma \vdash Q_1 \mid P_3' \triangleright \Delta_0 \cdot s_1[3] : [1]?(U);[2]?(U);\mathtt{end} \approx^s \mathbf{0} \triangleright s_1[1] : \mathtt{end} \cdot s_1[2] : \mathtt{end} \cdot s_1[3] : \mathtt{end}$$

since $(\Gamma,\Delta) \not\xrightarrow{\ell}$ for any $\ell \neq \tau$ with $\Delta = \Delta_0 \cdot s_1[3] : [1]?(U);[2]?(U);\mathtt{end}$ (by the condition of Line 3 in Figure 2). However by the untyped LTS, $Q_1 \mid P_3' \not\approx \mathbf{0}$ since $Q_1 \mid P_3' \xrightarrow{s_1[1][3]!\langle v\rangle}$.

4 Globally Governed Behavioural Theory

We introduce the semantics for globally governed behavioural theory. In the previous section, the local typing (Δ) constrains the untyped LTS to give rise to a local typed LTS. In a multiparty distributed environment, communications follow the global protocol, which controls both an observed process and its observer. The local typing is not sufficient to maintain the consistency of transitions of a process with respect to a global protocol. In this section we refine the environment LTS with a *global environment* E to give a more fine-grained control over the LTS of the processes.

Global Environments and Configurations. We define a *global environment* $(E, E', \ldots)$ as a mapping from session names to global types.

$$E \ ::= \ E \cdot s : G \ \mid \ \emptyset$$

The projection definition is extended to include E as $\mathtt{proj}(E) = \bigcup_{s:G \in E} \mathtt{proj}(s : G)$.

We define a labelled reduction relation over global environments which corresponds to $\Delta \longrightarrow \Delta'$ defined in § 2. We use the labels $\lambda \in \{s : \mathtt{p} \to \mathtt{q} : U, s : \mathtt{p} \to \mathtt{q} : l\}$ to annotate reductions over global environments. We define $\mathtt{out}(\lambda)$ and $\mathtt{inp}(\lambda)$ as $\mathtt{out}(s : \mathtt{p} \to \mathtt{q} : U) = \mathtt{out}(s : \mathtt{p} \to \mathtt{q} : l) = \mathtt{p}$ and as $\mathtt{inp}(s : \mathtt{p} \to \mathtt{q} : U) = \mathtt{inp}(s : \mathtt{p} \to \mathtt{q} : l) = \mathtt{q}$ and $\mathtt{p} \in \ell$ if $\mathtt{p} \in \mathtt{out}(\ell) \cup \mathtt{inp}(\ell)$. We often omit the label λ by writing $\longrightarrow$ for $\xrightarrow{\lambda}$ and $\longrightarrow^*$ for $(\xrightarrow{\lambda})^*$. The first rule is the axiom for the input and output interaction between two parties; the second rule is for the choice; the third and forth rules formulate the case that the action λ can be performed under $\mathtt{p} \to \mathtt{q}$ if $\mathtt{p}$ and $\mathtt{q}$ are not related to the participants in λ; and the fifth rule is a congruent rule.

$$\{s : \mathtt{p} \to \mathtt{q} : \langle U\rangle.G\} \xrightarrow{s:\mathtt{p}\to\mathtt{q}:U} \{s : G\} \qquad \{s : \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i\in I}\} \xrightarrow{s:\mathtt{p}\to\mathtt{q}:l_k} \{s : G_k\}$$

$$\frac{\{s:G\} \xrightarrow{\lambda} \{s:G'\} \quad \mathtt{p},\mathtt{q} \notin \lambda}{\{s:\mathtt{p}\to\mathtt{q}:\langle U\rangle.G\} \xrightarrow{\lambda} \{s:\mathtt{p}\to\mathtt{q}:\langle U\rangle.G'\}}$$

$$\frac{\{s:G_i\} \xrightarrow{\lambda} \{s:G_i'\} \quad i\in I, \quad \mathtt{p},\mathtt{q} \notin \lambda}{\{s:\mathtt{p}\to\mathtt{q}:\{l_i:G_i\}_{i\in I}\} \xrightarrow{\lambda} \{s:\mathtt{p}\to\mathtt{q}:\{l_i:G_i'\}_{i\in I}\}} \qquad \frac{E \xrightarrow{\lambda} E'}{E\cdot E_0 \xrightarrow{\lambda} E'\cdot E_0}$$

As a simple example of the above LTS, consider $s:\mathtt{p}\to\mathtt{q}:\langle U_1\rangle.\mathtt{p}'\to\mathtt{q}':\{l_1:\mathtt{end}, l_2:\mathtt{p}'\to\mathtt{q}':\langle U_2\rangle.\mathtt{end}\}$. Since $\mathtt{p},\mathtt{q},\mathtt{p}',\mathtt{q}'$ are pairwise distinct, we can apply the second and third rules to obtain: $s:\mathtt{p}\to\mathtt{q}:\langle U_1\rangle.\mathtt{p}'\to\mathtt{q}':\{l_1:\mathtt{end}, l_2:\mathtt{p}'\to\mathtt{q}':\langle U_2\rangle.\mathtt{end}\} \xrightarrow{s:\mathtt{p}'\to\mathtt{q}':l_1} s:\mathtt{p}\to\mathtt{q}:\langle U_1\rangle.\mathtt{end}$

Next we introduce the *governance judgement* which controls the behaviour of processes by the global environment.

Definition 4.1 (Governance judgement). *Let $\Gamma \vdash P \triangleright \Delta$ be coherent. We write $E, \Gamma \vdash P \triangleright \Delta$ if $\exists E' \cdot E \longrightarrow^* E'$ and $\Delta \subseteq \mathit{proj}(E')$.*

The global environment E records the knowledge of both the environment (Δ) of the observed process P and the environment of its *observer*. The side conditions ensure that E is coherent with Δ: there exist E' reduced from E whose projection should cover the environment of P (since E should include the observer's information together with the observed process information recorded into Δ).

Next we define the LTS for well-formed environment configurations.

Definition 4.2 (Environment configuration). *We write (E,Γ,Δ) if $\exists E' \cdot E \longrightarrow^* E'$ and $\Delta \subseteq \mathit{proj}(E')$.*

Figure 3 defines a LTS over environment configurations that refines the LTS over environments (i.e $(\Gamma,\Delta) \xrightarrow{\ell} (\Gamma',\Delta')$) in § 3.

Each rule requires a corresponding environment transition (Figure 2 in § 3) and a corresponding labelled global environment transition in order to control a transition following the global protocol. [Acc] is the rule for accepting a session initialisation so that it creates a new mapping $s:G$ which matches Γ in a governed environment E. [Req] is the rule for requesting a new session and it is dual to [Acc].

The next seven rules are the transition relations on session channels and we assume the condition $\mathtt{proj}(E_1) \supseteq \Delta$ to ensure the base action of the environment matches one in a global environment. [Out] is a rule for the output where the type of the value and the action of (Γ,Δ) meets those in E. [In] is a rule for the input and dual to [Out]. [ResN] is a scope opening rule for a name so that the environment can perform the corresponding type $\langle G\rangle$ of a. [ResS] is a scope opening rule for a session channel which creates a set of mappings for the opened session channel s' corresponding to the LTS of the environment. [Sel] and [Bra] are the rules for selection and branching, which is similar to [Out] and [In]. In [Tau] rule, we refined the reduction relation on Δ in § 2 as:

1. $\{s[\mathtt{p}]:[\mathtt{q}]!\langle U\rangle;T\cdot s[\mathtt{q}]:[\mathtt{p}]?(U);T'\} \xrightarrow{s:\mathtt{p}\to\mathtt{q}:U} \{s[\mathtt{p}]:T\cdot s[\mathtt{q}]:T'\}$.
2. $\{s[\mathtt{p}]:[\mathtt{q}]\oplus\{l_i:T_i\}_{i\in I}\cdot s[\mathtt{q}]:[\mathtt{p}]\&\{l_j:T_j'\}_{j\in J}\} \xrightarrow{s:\mathtt{p}\to\mathtt{q}:l_k} \{s[\mathtt{p}]:T_k\cdot s[\mathtt{q}]:T_k'\}\ I\subseteq J, k\in I$.
3. $\Delta\cup\Delta' \xrightarrow{\lambda} \Delta\cup\Delta''$ if $\Delta' \xrightarrow{\lambda} \Delta''$.

$$[\text{Acc}]\frac{\Gamma \vdash a : \langle G\rangle \quad (\Gamma,\Delta_1) \xrightarrow{a[A](s)} (\Gamma,\Delta_2)}{(E,\Gamma,\Delta_1) \xrightarrow{a[A](s)} (E\cdot s : G,\Gamma,\Delta_2)} \qquad [\text{Req}]\frac{\Gamma \vdash a : \langle G\rangle \quad (\Gamma,\Delta_1) \xrightarrow{\overline{a}[A](s)} (\Gamma,\Delta_2)}{(E,\Gamma,\Delta_1) \xrightarrow{\overline{a}[A](s)} (E\cdot s : G,\Gamma,\Delta_2)}$$

$$[\text{Out}]\frac{\Gamma \vdash v : U \quad (\Gamma,\Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle v\rangle} (\Gamma,\Delta_2) \quad E_1 \xrightarrow{s:\mathtt{p}\to\mathtt{q}:U} E_2}{(E_1,\Gamma,\Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle v\rangle} (E_2,\Gamma,\Delta_2)} \qquad [\text{In}]\frac{(\Gamma,\Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]?\langle v\rangle} (\Gamma\cdot v : U,\Delta_2) \quad E_1 \xrightarrow{s:\mathtt{q}\to\mathtt{p}:U} E_2}{(E_1,\Gamma,\Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]?\langle v\rangle} (E_2,\Gamma\cdot v : U,\Delta_2)}$$

$$[\text{ResN}]\frac{(\Gamma,\Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle a\rangle} (\Gamma\cdot a : \langle G\rangle,\Delta_2) \quad E_1 \xrightarrow{s:\mathtt{q}\to\mathtt{p}:\langle G\rangle} E_2}{(E_1,\Gamma,\Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle a\rangle} (E_2,\Gamma\cdot a : \langle G\rangle,\Delta_2)} \qquad [\text{ResS}]\frac{(\Gamma,\Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle s'[\mathtt{p}']\rangle} (\Gamma,\Delta_2\cdot\{s'[\mathtt{p}_i] : T_i\}) \quad E_1 \xrightarrow{s:\mathtt{q}\to\mathtt{p}:T} E_2 \quad \cdot \forall i.G\lceil \mathtt{p}_i = T_i}{(E_1,\Gamma,\Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle s'[\mathtt{p}']\rangle} (E_2\cdot s' : G,\Gamma,\Delta_2\cdot\{s'[\mathtt{p}_i] : T_i\})}$$

$$[\text{Sel}]\frac{(\Gamma,\Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]\oplus l} (\Gamma,\Delta_2) \quad E_1 \xrightarrow{s:\mathtt{p}\to\mathtt{q}:l} E_2}{(E_1,\Gamma,\Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]\oplus l} (E_2,\Gamma,\Delta_2)} \qquad [\text{Bra}]\frac{(\Gamma,\Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]\&l} (\Gamma,\Delta_2) \quad E_1 \xrightarrow{s:\mathtt{q}\to\mathtt{p}:l} E_2}{(E_1,\Gamma,\Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]\&l} (E_2,\Gamma,\Delta_2)}$$

$$[\text{Tau}]\frac{(\Delta_1 = \Delta_2,\ E_1 = E_2) \vee (\Delta_1 \xrightarrow{\lambda} \Delta_2,\ E_1 \xrightarrow{\lambda} E_2)}{(E_1,\Gamma,\Delta_1) \xrightarrow{\tau} (E_2,\Gamma,\Delta_2)} \qquad [\text{Inv}]\frac{E_1 \longrightarrow^* E_1' \quad (E_1',\Gamma_1,\Delta_1) \xrightarrow{\ell} (E_2,\Gamma_2,\Delta_2)}{(E_1,\Gamma_1,\Delta_1) \xrightarrow{\ell} (E_2,\Gamma_2,\Delta_2)}$$

Fig. 3. The LTS for the environment configuations

[Inv] is the key rule: the global environment E_1 reduces to E_1' to perform the observer's actions, hence the observed process can perform the action w.r.t. E_1'. Hereafter we write $\longrightarrow$ for $\xrightarrow{\tau}$.

Example 4.1 (LTS for environment configuration). Let $E = s : \mathtt{p} \to \mathtt{q} : \langle U\rangle.\mathtt{p} \to \mathtt{q} : \langle U\rangle.G$, $\Gamma = v : U$ and $\Delta = s[\mathtt{p}] : [\mathtt{q}]!\langle U\rangle; T_\mathtt{p}$ with $G\lceil \mathtt{p} = T_\mathtt{p}$, $G\lceil \mathtt{q} = T_\mathtt{q}$ and $\mathtt{roles}(G) = \{\mathtt{p},\mathtt{q}\}$. Then (E,Γ,Δ) is an environment configuration since if $E \longrightarrow E'$ then $\mathtt{proj}(E') \supset \Delta$ because $E \xrightarrow{s:\mathtt{p}\to\mathtt{q}:U} s : \mathtt{p} \to \mathtt{q} : \langle U\rangle.G$, $\mathtt{proj}(s : \mathtt{p} \to \mathtt{q} : \langle U\rangle.G) = s[\mathtt{p}] : [\mathtt{q}]!\langle U\rangle; T_\mathtt{p} \cdot s[\mathtt{q}] : [\mathtt{p}]?(U); T_\mathtt{q}$ and $\mathtt{proj}(s : \mathtt{p} \to \mathtt{q} : \langle U\rangle.G) \supset \Delta$. Then we can apply [Out] to $s : \mathtt{p} \to \mathtt{q} : \langle U\rangle.G \xrightarrow{s:\mathtt{p}\to\mathtt{q}:U} s : G$ and $(\Gamma, s[\mathtt{p}] : [\mathtt{q}]!\langle U\rangle; T_\mathtt{p}) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle v\rangle} (\Gamma, s[\mathtt{p}] : T_\mathtt{p})$ to obtain $(s : \mathtt{p} \to \mathtt{q} : \langle U\rangle.G,\Gamma,\Delta) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle v\rangle} (s : G,\Gamma, s[\mathtt{p}] : T_\mathtt{p})$. By this and $E \longrightarrow s : \mathtt{p} \to \mathtt{q} : \langle U\rangle.G$, using [Inv], we can obtain $(E,\Gamma,\Delta) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle v\rangle} (s : G,\Gamma, s[\mathtt{p}] : T_\mathtt{p})$, as required.

Governed Reduction-Closed Congruency. To define the reduction-closed congruency, we first refine the barb, which is controlled by the global witness where observables of a configuration are defined with the global environment of the observer.

$$\begin{array}{rl}(E,\Gamma,\Delta\cdot s[\mathtt{p}] : [\mathtt{q}]!\langle U\rangle; T) \downarrow_{s[\mathtt{p}][\mathtt{q}]} & \text{if } s[\mathtt{q}] \notin \mathtt{dom}(\Delta), \exists E'\cdot E \longrightarrow^* E' \xrightarrow{s:\mathtt{p}\to\mathtt{q}:U}, \Delta \subseteq \mathtt{proj}(E')\\ (E,\Gamma,\Delta\cdot s[\mathtt{p}] : [\mathtt{q}] \oplus \{l_i : T_i\}_{i\in I}) \downarrow_{s[\mathtt{p}][\mathtt{q}]} & \text{if } s[\mathtt{q}] \notin \mathtt{dom}(\Delta), \exists E'\cdot E \longrightarrow^* E' \xrightarrow{s:\mathtt{p}\to\mathtt{q}:l_k}, k \in I, \Delta \subseteq \mathtt{proj}(E'),\\ (E,\Gamma,\Delta) \downarrow_a & \text{if } a \in \mathtt{dom}(\Gamma)\end{array}$$

We write $(\Gamma,\Delta,E) \Downarrow_m$ if $(\Gamma,\Delta,E) \longrightarrow^* (\Gamma,\Delta',E')$ and $(\Gamma,\Delta',E') \downarrow_m$.

Let us write $T_1 \sqsubseteq T_2$ if the syntax tree of T_2 includes T_1. For example, $[\mathtt{q}]?(U'); T \sqsubseteq [\mathtt{p}]!\langle U\rangle; [\mathtt{q}]?(U'); T$. Then we define: $E_1 \sqcup E_2 = \{E_i(s) \mid E_j(s) \sqsubseteq E_i(s), i,j \in \{1,2\}, i \neq j\} \cup E_1 \setminus \mathtt{dom}(E_2) \cup E_2 \setminus \mathtt{dom}(E_1)$. As an example of $E_1 \sqcup E_2$, let us define:

$$\begin{array}{l}E_1 = s_1 : \mathtt{p} \to \mathtt{q} : \langle U_1\rangle.\mathtt{p}' \to \mathtt{q}' : \langle U_2\rangle.\mathtt{p} \to \mathtt{q} : \langle U_3\rangle.\mathtt{end} \cdot s_2 : \mathtt{p} \to \mathtt{q} : \langle W_2\rangle.\mathtt{end}\\ E_2 = s_1 : \mathtt{p} \to \mathtt{q} : \langle U_3\rangle.\mathtt{end} \cdot s_2 : \mathtt{p}' \to \mathtt{q}' : \langle W_1\rangle.\mathtt{p} \to \mathtt{q} : \langle W_2\rangle.\mathtt{end}\end{array}$$

Then $E_1 \sqcup E_2 = \mathtt{p} \to \mathtt{q} : \langle U_1 \rangle . \mathtt{p}' \to \mathtt{q}' : \langle U_2 \rangle . \mathtt{p} \to \mathtt{q} : \langle U_3 \rangle . \mathtt{end} \cdot s_2 : \mathtt{p}' \to \mathtt{q}' : \langle W_1 \rangle . \mathtt{p} \to \mathtt{q} : \langle W_2 \rangle . \mathtt{end}$. The behavioural relation w.r.t. a global whiteness is defined below.

Definition 4.3 (Configuration relation). The relation $\mathscr{R}$ is a *configuration relation* between two configurations $E_1, \Gamma \vdash P_1 \triangleright \Delta_1$ and $E_2, \Gamma \vdash P_2 \triangleright \Delta_2$, written $E_1 \sqcup E_2, \Gamma \vdash P \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$ if $E_1 \sqcup E_2$ is defined.

Proposition 4.1 (Decidability). *(1) Given E_1 and E_2, a problem whether $E_1 \sqcup E_2$ is defined or not is decidable and if it is defined, the calculation of $E_1 \sqcup E_2$ terminates; and (2) Given E, a set $\{E' \mid E \longrightarrow^* E'\}$ is finite.*

Definition 4.4 (Global configuration transition). We write $E_1, \Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} E_2, \Gamma' \vdash P_2 \triangleright \Delta_2$ if $E_1, \Gamma \vdash P_1 \triangleright \Delta_1$, $P_1 \xrightarrow{\ell} P_2$ and $(E_1, \Gamma, \Delta_1) \xrightarrow{\ell} (E_2, \Gamma', \Delta_2)$.

Proposition 4.2. *(1) $(E_1, \Gamma, \Delta_1) \xrightarrow{\ell} (E_2, \Gamma_2, \Delta_2)$ implies that $(E_2, \Gamma_2, \Delta_2)$ is an environment configuration; and (2) If $\Gamma \vdash P \triangleright \Delta$ and $P \longrightarrow P'$ with $\mathtt{co}(\Delta)$, then $E, \Gamma \vdash P \triangleright \Delta \longrightarrow E, \Gamma \vdash P' \triangleright \Delta'$ and $\mathtt{co}(\Delta')$.*

The definition of the reduction congruence for governance follows. Below we define $E, \Gamma \vdash P \triangleright \Delta \Downarrow_n$ if $P \Downarrow_m$ and $(E, \Gamma, \Delta) \Downarrow_m$.

Definition 4.5 (Governed reduction congruence). A configuration relation $\mathscr{R}$ is *governed reduction congruence* if $E, \Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$ then

1. $E, \Gamma \vdash P_1 \triangleright \Delta_1 \Downarrow_n$ if and only if $E, \Gamma \vdash P_2 \triangleright \Delta_2 \Downarrow_n$
2. $P_1 \twoheadrightarrow P_1'$ if and only if $P_2 \twoheadrightarrow P_2'$ and $E, \Gamma \vdash P_1' \triangleright \Delta_1' \mathscr{R} P_2' \triangleright \Delta_2'$
3. For all closed context C, such that $E, \Gamma \vdash C[P_1] \triangleright \Delta_1'$ and $E, \Gamma \vdash C[P_2] \triangleright \Delta_2'$ then $E, \Gamma \vdash C[P_1] \triangleright \Delta_1' \mathscr{R} C[P_2] \triangleright \Delta_2'$.

The union of all governed reduction congruence relations is denoted as $\cong_g^s$.

Globally Governed Bisimulation and Its Properties. This subsection introduces the globally governed bisimulation relation definition and studies its main properties.

Definition 4.6 (Globally governed bisimulation). A configuration relation $\mathscr{R}$ is a *globally governed weak bisimulation* (or governed bisimulation) if whenever $E, \Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$, it holds:

1. $E, \Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} E_1', \Gamma' \vdash P_1' \triangleright \Delta_1'$ implies $E, \Gamma \vdash P_2 \triangleright \Delta_2 \stackrel{\hat{\ell}}{\Longrightarrow} E_2', \Gamma' \vdash P_2' \triangleright \Delta_2'$ such that $E_1' \sqcup E_2', \Gamma' \vdash P_1' \triangleright \Delta_1' \mathscr{R} P_2' \triangleright \Delta_2'$.
2. The symmetric case.

The maximum bisimulation exists which we call *governed bisimilarity*, denoted by $\approx_g^s$. We sometimes leave environments implicit, writing e.g. $P \approx_g^s Q$.

Theorem 4.1 (Sound and completeness). $\approx_g^s = \cong_g^s$.

The relationship between $\approx^s$ and $\approx_g^s$ is given as follows.

Theorem 4.2. *If for all E, $E, \Gamma \vdash P_1 \triangleright \Delta_1 \approx_g^s P_2 \triangleright \Delta_2$ then $\Gamma \vdash P_1 \triangleright \Delta_1 \approx^s P_2 \triangleright \Delta_2$. Also if $\Gamma \vdash P_1 \triangleright \Delta_1 \approx^s P_2 \triangleright \Delta_2$, then for all E, $E, \Gamma \vdash P_1 \triangleright \Delta_1 \approx_g^s P_2 \triangleright \Delta_2$.*

To justify the above theorem, consider the following processes:

$$P_1 = s_1[1][3]!\langle v\rangle;s_2[1][2]!\langle w\rangle;\mathbf{0} \mid s_1[2][3]!\langle v\rangle;s_2[2][1]?(x);s_2[2][3]!\langle x\rangle;\mathbf{0}$$
$$P_2 = s_1[1][3]!\langle v\rangle;\mathbf{0} \mid s_2[1][2]!\langle w\rangle;\mathbf{0} \mid s_1[2][3]!\langle v\rangle;s_2[2][1]?(x);s_2[2][3]!\langle x\rangle;\mathbf{0}$$

then we have $P_1 \approx^s P_2$. By the above theorem, we expect that for all E, we have $E, \Gamma \vdash P_1 \triangleright \Delta_1$ and $E, \Gamma \vdash P_2 \triangleright \Delta_2$ then $E \vdash P_1 \approx_g^s P_2$. This is in fact true because the possible E that can type P_1 and P_2 are:

$$E_1 = s_1 : 1 \to 3 : \langle U\rangle.2 \to 3 : \langle U\rangle.\mathtt{end} \cdot s_2 : 1 \to 2 : \langle W\rangle.2 \to 3 : \langle W\rangle.\mathtt{end}$$
$$E_2 = s_1 : 2 \to 3 : \langle U\rangle.1 \to 3 : \langle U\rangle.\mathtt{end} \cdot s_2 : 1 \to 2 : \langle W\rangle.2 \to 3 : \langle W\rangle.\mathtt{end}$$

Note that all E that are instances up-to weakening are E_1 and E_2.

To clarify the difference between $\approx^s$ and $\approx_g^s$, we introduce the notion of a *simple multiparty process* defined in [11]. A simple process contains only a single session so that it satisfies the progress property as proved in [11]. Formally a process P is *simple* when it is typable with a type derivation where the session typing in the premise and the conclusion of each prefix rule is restricted to at most a single session (i.e. any $\Gamma \vdash P \triangleright \Delta$ which appears in a derivation, Δ contains at most one session channel in its domain, see [11]). Since there is no interleaving of sessions in simple processes, the difference between $\approx^s$ and $\approx_g^s$ disappears.

Theorem 4.3 (Coincidence). *Assume P_1 and P_2 are simple. If $\exists E \cdot E, \Gamma \vdash P_1 \triangleright \Delta_1 \approx_g^s P_2 \triangleright \Delta_2$ then $\Gamma \vdash P_1 \triangleright \Delta_1 \approx^s P_2 \triangleright \Delta_2$.*

To justify the above theorem, consider: $P_1 = s[1][2]?(x);s[1][3]!\langle x\rangle;\mathbf{0} \mid s[2][1]!\langle v\rangle;\mathbf{0}$ and $P_2 = s[1][3]!\langle v\rangle;\mathbf{0}$. It holds that for $E = s : 2 \to 1 : \langle U\rangle.1 \to 3 : \langle U\rangle.\mathtt{end}$ then $E \vdash P_1 \approx_g^s P_2$. We can easily reason $P_1 \approx^s P_2$.

Example 4.2 (Governed bisimulation). Recall the example from § 1 and Example 3.1. Q_1 is the process corresponding to Example 3.1, while Q_2 has a parallel thread instead of the sequential composition (this corresponds to $P_1 \mid R_2$ in § 1).

$$Q_1 = s_1[1][3]!\langle v\rangle;s_2[1][2]!\langle w\rangle;\mathbf{0} \mid s_2[2][1]?(x);\mathbf{0} \mid s_1[2][3]!\langle v\rangle;\mathbf{0}$$
$$Q_2 = s_1[1][3]!\langle v\rangle;s_2[1][2]!\langle w\rangle;\mathbf{0} \mid s_2[2][1]?(x);s_1[2][3]!\langle v\rangle;\mathbf{0}$$

Assume: $\Gamma = v : S \cdot w : S$
$$\Delta = s_1[1] : [3]!\langle S\rangle;\mathtt{end} \cdot s_1[2] : [3]!\langle S\rangle;\mathtt{end} \cdot s_2[1] : [2]!\langle S\rangle;\mathtt{end} \cdot s_2[2] : [1]?(S);\mathtt{end}$$

Then we have $\Gamma \vdash Q_1 \triangleright \Delta$ and $\Gamma \vdash Q_2 \triangleright \Delta$. Now assume the two global witnesses as:

$$E_1 = s_1 : 1 \to 3 : \langle S\rangle.2 \to 3 : \langle S\rangle.\mathtt{end} \cdot s_2 : 1 \to 2 : \langle S\rangle.\mathtt{end}$$
$$E_2 = s_1 : 2 \to 3 : \langle S\rangle.1 \to 3 : \langle S\rangle.\mathtt{end} \cdot s_2 : 1 \to 2 : \langle S\rangle.\mathtt{end}$$

Then the projection of E_1 and E_2 are given as:

$$\begin{aligned}\mathtt{proj}(E_1) = {} & s_1[1] : [3]!\langle S\rangle;\mathtt{end} \cdot s_1[2] : [3]!\langle S\rangle;\mathtt{end} \cdot s_1[3] : [1]?(S);[2]?(S);\mathtt{end} \\ & s_2[1] : [2]!\langle S\rangle;\mathtt{end} \cdot s_2[2] : [1]?(S);\mathtt{end} \\ \mathtt{proj}(E_2) = {} & s_1[1] : [3]!\langle S\rangle;\mathtt{end} \cdot s_1[2] : [3]!\langle S\rangle;\mathtt{end} \cdot s_1[3] : [2]?(S);[1]?(S);\mathtt{end} \cdot \\ & s_2[1] : [2]!\langle S\rangle;\mathtt{end} \cdot s_2[2] : [1]?(S);\mathtt{end}\end{aligned}$$

with $\Delta \subset \mathtt{proj}(E_1)$ and $\Delta \subset \mathtt{proj}(E_2)$. The reader should note that the difference between E_1 and E_2 is the type of the participant 3 at s_1.

By definition, we can write: $E_i, \Gamma \vdash Q_1 \triangleright \Delta$ and $E_i, \Gamma \vdash Q_2 \triangleright \Delta$ for $i = 1,2$. Both processes are well-formed global configurations under both witnesses. Now we can observe $\Gamma \vdash Q_1 \triangleright \Delta \overset{s[2][3]!\langle v\rangle}{\longrightarrow} \Gamma \vdash Q_1' \triangleright \Delta'$ but $\Gamma \vdash Q_2 \triangleright \Delta \overset{s[2][3]!\langle v\rangle}{\not\longrightarrow}$. Hence $\Gamma \vdash Q_1 \triangleright \Delta \not\approx^s Q_2 \triangleright \Delta$. By the same argument, we have: $E_2, \Gamma \vdash Q_1 \triangleright \Delta \not\approx_g^s Q_2 \triangleright \Delta$. On the other hand, since E_1 *forces* to wait for $s[2][3]!\langle v\rangle$, $E_1, \Gamma \vdash Q_1 \triangleright \Delta \overset{s[2][3]!\langle v\rangle}{\not\longrightarrow}$. Hence Q_1 and Q_2 are bisimilar, i.e. $E_1, \Gamma \vdash Q_1 \triangleright \Delta \approx_g^s Q_2 \triangleright \Delta$. This concludes the optimisation is correct.

5 Related and Future Work

As a typed foundation for structured communications programming, session types [9, 17] have been studied over the last decade for a wide range of process calculi and programming languages. Recently several works developed multiparty session types and their extensions. While typed behavioural equivalences are one of the central topics of the π-calculus, surprisingly the typed behavioural semantics based on session types have been less explored, and the existing ones only focus on binary (two-party) sessions. Our work [14] develops an *asynchronous binary* session typed behavioural theory with event operations. An LTS is defined on session type process judgements and ensures session typed properties, such as linearity in the presence of asynchronous queues. The work [15] proves the proof conversions induced by Linear Logic interpretation coincide with an observational equivalence over a strict subset of the binary synchronous session processes. The main focus of our paper is *multiparty* session types and governed bisimulation, whose definitions and properties crucially depend on information of global types. In the first author's PhD thesis [13], we studied how governed bisimulations can be systematically developed under various semantics including three kinds of asynchronous semantics by modularly changing the LTS for processes, environments and global types. For governed bisimulations, we can reuse all of the definitions among four semantics by only changing the conditions of the LTS of global types to suit each semantics. Another recent work [6] gives a fully abstract encoding of a *binary* synchronous session typed calculus into a linearly typed π-calculus [3]. We believe the same encoding method is smoothly applicable to $\approx^s$ since it is defined solely based on the projected types (i.e. local types). However a governed bisimulation requires a global witness, hence the additional global information would be required for full abstraction.

The constructions of our work are hinted by [8] which studies typed behavioural semantics for the π-calculus with IO-subtyping where a LTS for pairs of typing environments and processes is used for defining typed testing equivalences and barbed congruence. On the other hand, in [8], the type environment indexing the observational equivalence resembles more a dictator where the refinement can be obtained by the fact that the observer has only partial knowledge on the typings, than a coordinator like our approach. Several papers have developed bisimulations for the higher-order π-calculus or its variants using the information of the environments. Among them, a recent paper [12] uses a pair of a process and an observer knowledge set for the LTS. The knowledge set contains a mapping from first order values to the higher-order processes, which allows a tractable higher-order behavioural theory using the first-order LTS.

We record a choreographic type as the witness in the environment to obtain fine-grained bisimulations of multiparty processes. The highlight of our bisimulation

construction is an effective use of the semantics of global types for LTSs of processes (cf. [Inv] in Figure 3 and Definition 4.4). Global types can give a guidance how to coordinate parallel threads giving explicit protocols, hence it is applicable to a semantic-preserving optimisation (cf. Example 4.2 and [7]). While it is known that it is undecidable to check $P \approx Q$ in the full π-calculus, it is an interesting future topic to investigate automated bisimulation-checking techniques for the governed bisimulations for some subset of multiparty session processes.

Acknowledgement. The work has been partially sponsored by the Ocean Observatories Initiative and EPSRC EP/K011715/1, EP/K034413/1 and EP/G015635/1.

References

1. Ocean Observatories Initiative (OOI), `http://www.oceanobservatories.org/`
2. Amadio, R.M., Castellani, I., Sangiorgi, D.: On bisimulations for the asynchronous pi-calculus. TCS 195(2), 291–324 (1998)
3. Berger, M., Honda, K., Yoshida, N.: Sequentiality and the π-calculus. In: Abramsky, S. (ed.) TLCA 2001. LNCS, vol. 2044, pp. 29–45. Springer, Heidelberg (2001)
4. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008)
5. W3C Web Services Choreography, `http://www.w3.org/2002/ws/chor/`
6. Demangeon, R., Honda, K.: Full abstraction in a subtyped pi-calculus with linear types. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 280–296. Springer, Heidelberg (2011)
7. Technical report of this paper. Department of Computing, Imperial College, DTR 2013/4
8. Hennessy, M., Rathke, J.: Typed behavioural equivalences for processes in the presence of subtyping. Mathematical Structures in Computer Science 14(5), 651–684 (2004)
9. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type disciplines for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
10. Honda, K., Yoshida, N.: On reduction-based process semantics. TCS 151(2), 437–486 (1995)
11. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: POPL 2008, pp. 273–284. ACM (2008)
12. Koutavas, V., Hennessy, M.: A testing theory for a higher-order cryptographic language. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 358–377. Springer, Heidelberg (2011)
13. Kouzapas, D.: A study of bisimulation theory for session types. PhD thesis, Department of Computing, Imperial College London (May 2013)
14. Kouzapas, D., Yoshida, N., Honda, K.: On asynchronous session semantics. In: Bruni, R., Dingel, J. (eds.) FORTE 2011 and FMOODS 2011. LNCS, vol. 6722, pp. 228–243. Springer, Heidelberg (2011)
15. Pérez, J.A., Caires, L., Pfenning, F., Toninho, B.: Linear logical relations for session-based concurrency. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 539–558. Springer, Heidelberg (2012)
16. Pierce, B., Sangiorgi, D.: Typing and subtyping for mobile processes. MSCS 6(5), 409–454 (1996)
17. 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)
18. Yoshida, N.: Graph types for monadic mobile processes. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 371–386. Springer, Heidelberg (1996)

A General Proof System for Modalities in Concurrent Constraint Programming

Vivek Nigam[1], Carlos Olarte[2], and Elaine Pimentel[3]

[1] Universidade Federal da Paraíba, Brazil
[2] Pontificia Universidad Javeriana-Cali, Colombia
[3] Universidade Federal de Minas Gerais, Brazil

Abstract. The combination of timed, spatial, and epistemic information is often needed in the specification of modern concurrent systems. We propose the proof system SELL$^{\cap}$, which extends linear logic with subexponentials with quantifiers over subexponentials, therefore allowing for an arbitrary number of modalities. We then show how a proper structure of the subexponential signature in SELL$^{\cap}$ allows for the specification of concurrent systems with timed, spatial, and epistemic modalities. In the context of Concurrent Constraint Programming (CCP), a declarative model of concurrency, we illustrate how the view of subexponentials as specific modalities is general enough to modularly encode into SELL$^{\cap}$ variants of CCP with these three modalities, thus providing a proof-theoretic foundations for those calculi.

1 Introduction

To specify the behavior of distributed agents or the policies governing a distributed system, it is often necessary to reason by using different types of modalities, such as time, space, or even the epistemic state of agents. For instance, the access-control policies of a building might allow Bob to have access only in some pre-defined time, such as its opening hours. Another policy might also allow Bob to ask Alice who has higher credentials to grant him access to the building, or even specify that Bob has only access to some specific rooms of the building. Following this need, many formalisms have been proposed to specify, program and reason about such policies, *e.g.*, Ambient Calculus, Concurrent Constraint Programming, Authorization Logics, just to name a few.

Logic and proof theory have often inspired the design of many of these formalisms. For example, Saraswat *et al.* proposed Concurrent Constraint Programming (CCP), a model for concurrency that combines the traditional operational view of process calculi with a declarative view based on logic [16,15] (see [13] for a survey). Agents in CCP *interact* with each other by *telling* and *asking* information represented as *constraints* to a global store. Later, Fages *et al.* in [4] proposed Linear Concurrent Constraint (`lcc`), inspired by linear logic [6], to allow the use of linear constraints, that is, tokens of information that once used by an agent are removed from the global store.

In order to capture the behavior of distributed systems which take into account spatial, temporal and/or epistemic properties, new formalisms have been proposed. For instance, Saraswat *et al.* proposed `tcc` [17], which extends CCP with time modalities.

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 410–424, 2013.

More recently, Knight *et al.* [7] proposed a CCP-based language with spatial and epistemic modalities. Some of these developments have also been followed by a similar development in proof theory. For instance, Nigam proposed a framework for linear authorization logics [9], which allow the specification of access control policies that may mention the affirmations, possessions and knowledge of principals and demonstrated that a wide range of linear authorization policies can be specified in linear logic with subexponentials (*SELL*) [2,10].

This paper shows that timed, spatial, and epistemic modalities can be *uniformly* specified in a single logical framework called SELL$^{\Cap}$. Our first contribution is the introduction of the proof system SELL$^{\Cap}$, which extends *SELL* with universal ($\Cap$) and existential ($\Cup$) quantifiers over subexponentials. It turns out that SELL$^{\Cap}$ has good proof-theoretic properties: it admits cut-elimination and also has a complete focusing discipline [1].

For our second contribution, we show that subexponentials can be interpreted as spatial, epistemic and temporal modalities, thus providing a framework for specifying concurrent systems with these modalities. This is accomplished by encoding different CCP languages, for which the proposed quantifiers play an important role. For instance, they enable the use of an *arbitrary number of subexponentials*, required to model the unbounded nesting of modalities, which is a common feature in epistemic and spatial systems. This do not seem possible in existing logical frameworks such as [18] that do not contain subexponentials nor its quantifiers.

Another important feature of subexponentials is that they can be organized into a pre-order, which specifies the provability relation among them. By coupling subexponential quantifiers with a suitable pre-order, it is possible to specify *declaratively* the rules in which agents can manipulate information. For example, an agent cannot see the information contained in a space that she does not have access to. The boundaries are naturally implied by the pre-order of subexponentials.

This work opens a number of possibilities for specifying the behavior of distributed systems. For instance, unlike [7], it seems possible in our framework to handle an infinite number of agents. Moreover, we discuss how linearity of constraints can be straightforwardly included to these systems to represent, *e.g.*, agents that can *update*/*change* the content of the distributed spaces. Also, by changing the underlying subexponential structure, different modalities can be put in the hands of the modelers and programmers. Finally, all the linear logic meta-theory becomes available for reasoning about distributed systems featuring modalities.

Organization. In Section 2 we review the proof theory of *SELL*, identify its limitations, and propose an extension (SELL$^{\Cap}$) allowing for the quantification of subexponentials ($\Cap$ and $\Cup$). We prove that SELL$^{\Cap}$ admits cut-elimination. Section 3 reviews some background on CCP, for which we provide a sound and faithful encoding in SELL$^{\Cap}$. As we shall show, our encoding is modular enough to extend it so to specify new constructs involving modalities, namely, constructs for epistemic (Section 4.2), spatial (Section 4.3) and temporal modalities (Section 4.4). In Section 5 we identify a number of future work directions that we are currently working on. The detailed proofs and the focused presentation of SELL$^{\Cap}$ appear in the extended version of this paper, available at the authors' personal pages.

2 Linear Logic and Subexponential Quantifiers

Linear logic with subexponentials (*SELL*) shares with linear logic all connectives except the exponentials: the multiplicative and additive conjunctions, $\otimes$ and $\&$, linear implication, $\multimap$, additive disjunction $\oplus$, units $1, \bot, 0, \top$, and its universal and existential quantifiers $\forall$ and $\exists$. Their proof rules are the same as in standard linear logic [6]. However, instead of having a single pair of exponentials ! and ?, *SELL* may contain as many *labeled* exponentials ($!^l$ and $?^l$) as needed, called *subexponentials* [2,10].

Formally, the proof system for intuitionistic *SELL* is specified by a *subexponential signature* $\Sigma = \langle I, \preceq, U\rangle$, where I is a set of labels, $U \subseteq I$ is a set specifying which subexponentials allow weakening and contraction, and $\preceq$ is a pre-order among the elements of I. We assume that U is closed wrt $\preceq$, *i.e.*, if $a \in U$ and $a \preceq b$, then $b \in U$. The system *SELL* is constructed by adding all the rules for the linear logic connectives as usual, except for the exponentials, whose right introduction rules are as follows. For each $a \in I$, we add the introduction rules corresponding to dereliction and promotion, where we state explicitly the first-order signature $\mathcal{L}$ of the terms of the language:

$$\frac{\mathcal{L}; \Gamma, F \longrightarrow G}{\mathcal{L}; \Gamma, !^a F \longrightarrow G}\ !^a{}_L \quad \text{and} \quad \frac{\mathcal{L}; !^{x_1}F_1, \ldots !^{x_n}F_n \longrightarrow G}{\mathcal{L}; !^{x_1}F_1, \ldots !^{x_n}F_n \longrightarrow !^a G}\ !^a{}_R$$

The rules for $?^a$ are dual. Here, the rule $!^a{}_R$ (and $?^a{}_L$) have the side condition that $a \preceq x_i$ for all i. That is, one can only introduce a $!^a$ on the right (or a $?^a$ on the left) if all other formulas in the sequent are marked with indices that are greater or equal than a. Furthermore, for all $a \in U$, we add the structural rules:

$$\frac{\mathcal{L}; \Gamma, !^a F, !^a F \longrightarrow G}{\mathcal{L}; \Gamma, !^a F \longrightarrow G}\ C \quad \text{and} \quad \frac{\mathcal{L}; \Gamma \longrightarrow G}{\mathcal{L}; \Gamma, !^a F \longrightarrow G}\ W$$

That is, we are also free to specify which indices are *unbounded* (those appearing in the set U), and which indices are *linear* or *bounded*.

It is known that subexponentials greatly increase the expressiveness of the system when compared to linear logic. For instance, they can be used to represent contexts of proof systems [12], to mark the epistemic state of agents [9], or to specify locations in sequential computations [10].

The key difference to standard presentations of linear logic is that while linear logic has only seven logically distinct prefixes of bangs and question-marks, *SELL* allows for an unbounded number of such prefixes, *e.g.*, $!^i$, or $!^i?^j$. As we show later, by using different prefixes (written generically as $\bigtriangledown$), we will also be able to interpret subexponentials in more creative ways, such as temporal units or spatial and epistemic modalities.

However, *SELL* has a serious limitation: it does not have any sort of quantification over subexponentials. Therefore, given the interpretation above for subexponentials, it is not feasible in *SELL* to specify properties that are valid for all locations or for all agents. Another way of visualizing this limitation is that any sequent in any derivation in *SELL* has the same subexponential signature Σ. For instance, it is not possible to encode in *SELL* none of the encodings of the CCP languages discussed in Section 4.

2.1 Subexponential Quantifiers

In the following we introduce the system SELL$^{\Cap}$, containing two novel connectives: universal ($\Cap$) and existential ($\Cup$) quantifiers over *subexponentials*.

Subexponential Constants and Variables Recall that given a pre-order $(I, \preceq)$, the *ideal* of an element $a \in I$ in $\preceq$, written $\downarrow a$, is the set $\{x \mid x \preceq a\}$. The subexponential signature of SELL$^{\Cap}$ is of the form $\Sigma = \langle I, \preceq, F, U\rangle$, where I is a set of subexponential constants and $\preceq$ is a pre-order among these constants. The new component $F = \{\mathfrak{f}_1, \ldots, \mathfrak{f}_n\}$ specifies families of subexponentials indices. In particular, a family $\mathfrak{f} \in F$ takes an element of $a \in I$ and returns a subexponential index $\mathfrak{f}(a)$. As it will be clear below, families allow us to specify disjoint pre-orders based on $\langle I, \preceq\rangle$. The set of unbounded subexponentials $U \subseteq \{\mathfrak{f}(a) \mid a \in I, \mathfrak{f} \in F\}$, as before, is upwardly closed wrt $\preceq$: if $a \preceq b$, where $a, b \in I$, and $\mathfrak{f}(a) \in U$ then $\mathfrak{f}(b) \in U$. Notice that the SELL$^{\Cap}$ system obtained from the signature $\langle I, \preceq, \{id\}, U\rangle$ conservatively extends the *SELL* system obtained from $\langle I, \preceq, U\rangle$.

For our subexponential quantification, we will be interested in determining whether a subexponential b belongs to the ideal $\downarrow a$ of a given subexponential a. This is formally achieved by adding a typing information to subexponentials. Given the signature $\Sigma = \langle I, \preceq, F, U\rangle$, the judgment $s : a$ is true whenever $s \preceq a$. Thus we obtain the set $\mathcal{A}_\Sigma = \{s : a \mid s, a \in I, s \preceq a\}$ of typed *subexponential constants*. We shall simply write $!^{\mathfrak{f}(l)}$ instead of $!^{\mathfrak{f}(l\,:\,a)}$ when the type "a" can be inferred from the context. Similarly for "?."

As with the universal quantifier $\forall$, which introduces *eigenvariables* to the signature, the universal quantification for subexponentials $\Cap$ introduces *subexponential variables* of the shape $l : a$, where a is a subexponential constant, *i.e.*, $a \in I$. Thus, SELL$^{\Cap}$ sequents have the form $\mathcal{A}; \mathcal{L}; \Gamma \longrightarrow G$, where $\mathcal{A} = \mathcal{A}_\Sigma \cup \{l_1 : a_1, \ldots, l_n : a_n\}$, and $\{l_1, \ldots, l_n\}$ is a disjoint set of subexponential variables and $\{a_1, \ldots, a_n\} \subseteq I$ are subexponential constants. Formally, only these subexponential constants and variables may appear free as an index of subexponential bangs and question marks.

The introduction rules for the subexponential quantifiers look similar to those introducing the first-order quantifiers, but instead of manipulating the context $\mathcal{L}$, they manipulate the context $\mathcal{A}$:

$$\frac{\mathcal{A}; \mathcal{L}; \Gamma, P[l/x] \longrightarrow G}{\mathcal{A}; \mathcal{L}; \Gamma, \Cap x : a.P \longrightarrow G}\ \Cap_L \qquad \frac{\mathcal{A}, l_e : a; \mathcal{L}; \Gamma \longrightarrow G[l_e/x]}{\mathcal{A}; \mathcal{L}; \Gamma \longrightarrow \Cap x : a.G}\ \Cap_R$$

$$\frac{\mathcal{A}, l_e : a; \mathcal{L}; \Gamma, P[l_e/x] \longrightarrow G}{\mathcal{A}; \mathcal{L}; \Gamma, \Cup x : a.P \longrightarrow G}\ \Cup_L \qquad \frac{\mathcal{A}; \mathcal{L}; \Gamma \longrightarrow G[l/x]}{\mathcal{A}; \mathcal{L}; \Gamma \longrightarrow \Cup x : a.G}\ \Cup_R$$

In these rules, $l : a \in \mathcal{A}$ and l_e is fresh, *i.e.*, it does not appear in $\mathcal{A}$ nor $\mathcal{L}$. Intuitively, subexponential variables play a similar role as eigenvariables. The generic variable $l_i : a_i$ represents *any subexponential constant* that is in the ideal of the subexponential constant a. This is formalized by constructing from a given sequent, $\mathcal{A}; \mathcal{L}; \Gamma \longrightarrow G$, a pre-order, called *sequent pre-order*, written $\preceq_{\mathcal{A}}$. This pre-order is formally used in the side condition of the promotion rule and is defined on subexponentials obtained from applying a family $\mathfrak{f}_i \in F$ to an element of I. Formally, it is the *transitive* and *reflexive closure* of the following sets:

$$\{\mathfrak{f}(s_i : a) \preceq_{\mathcal{A}} \mathfrak{f}(s_j : b) \mid \mathfrak{f} \in F, s_i, s_j \in I \text{ and } s_i \preceq s_j\} \cup$$
$$\{\mathfrak{f}(l : a) \preceq_{\mathcal{A}} \mathfrak{f}(s : b) \mid \mathfrak{f} \in F, l \notin I, s \in I \text{ and } a \preceq s\}$$

The first component of this set specifies that families preserve the pre-order $\preceq$ in Σ only involving subexponential constants; thus $\preceq_{\mathcal{A}}$ is a conservative extension of $\preceq$. The second component is the interesting one, which relates subexponential obtained from variables and subexponentials obtained from constants: $l : a$ means that l belongs to the

ideal of a and if $a \preceq s$, then $\mathfrak{f}(l) \preceq_{\mathcal{A}} \mathfrak{f}(s)$. Notice that $\mathfrak{f}(l_1)$ and $\mathfrak{f}(l_2)$ are unrelated for two different subexponentials variables l_1 and l_2.

The pre-order $\preceq_{\mathcal{A}}$ is used in the right-introduction of bangs and the left-introduction of question-marks in a similar way as before in *SELL*

$$\frac{\mathcal{A}; \mathcal{L}; !^{\mathfrak{f}(l_1:\, a_1)}F_1, \ldots !^{\mathfrak{f}(l_n:\, a_n)}F_n \longrightarrow G}{\mathcal{A}; \mathcal{L}; !^{\mathfrak{f}(l_1:\, a_1)}F_1, \ldots, !^{\mathfrak{f}(l_n:\, a_n)}F_n \longrightarrow !^{\mathfrak{f}(l\,:\, a)}G}\ !^{\mathfrak{f}(l:a)}{}_R$$

$$\frac{\mathcal{A}; \mathcal{L}; !^{\mathfrak{f}(l_1:\, a_1)}F_1, \ldots !^{\mathfrak{f}(l_n:\, a_n)}F_n, P \longrightarrow ?^{\mathfrak{f}(l_{n+1}:\, a_{n+1})}G}{\mathcal{A}; \mathcal{L}; !^{\mathfrak{f}(l_1:\, a_1)}F_1, \ldots, !^{\mathfrak{f}(l_n:\, a_n)}F_n, ?^{\mathfrak{f}(l\,:\, a)}P \longrightarrow ?^{\mathfrak{f}(l_{n+1}:\, a_{n+1})}G}\ ?^{\mathfrak{f}(l:a)}{}_L$$

with the side condition that for all $1 \leq i \leq n+1$, $\mathfrak{f}(l : a) \preceq_{\mathcal{A}} \mathfrak{f}(l_i : a_i)$.

Notice that bangs and question marks use families, while quantifiers use only constants and variables. This interplay allows us to bind formulas with different families, such as in the formula $\Cap l : a.(!^{\mathfrak{f}(l\,:\, a)}F \otimes !^{\mathfrak{g}(l\,:\, a)}F')$.

As pointed out in [2], for cut-elimination, one needs to be careful with the structural properties of subexponentials. For subexponential variables, we define $\mathfrak{f}(l_i : a)$ to be always bounded, while for subexponential constants, it is similar as before: if $\mathfrak{f}(s : a) \in U$, then structural rules can be applied. We can now state our desired result.

Theorem 1. *For any signature Σ, the proof system SELL$^{\Cap}$ admits cut-elimination.*

3 CCP Calculi

Concurrent Constraint Programming (CCP) [15,16] is a model for concurrency that combines the traditional operational view of process calculi with a *declarative* view based on logic. This allows CCP to benefit from the large set of reasoning techniques of both process calculi and logic. In CCP, processes *interact* with each other by *telling* and *asking* constraints (pieces of information) in a common store of partial information. The type of constraints processes may act on is not fixed but parametric in a constraint system (CS for short). Such systems can be formalized as a Scott information system as in [15], or they can be built upon a suitable fragment of logic *e.g.*, as in [8]. Here we specify constraints as formulas in a fragment of intuitionistic first-order logic (LJ [5]).

Definition 1 (Constraint System [4]). *A constraint system is a tuple $(C, \vdash_\Delta)$ where C is a set of formulas (constraints) built from a first-order signature and the grammar*

$$F := 1 \mid A \mid F \wedge F \mid \exists \overline{x}.F$$

where A is an atomic formula. We shall use c, c', d, d', etc, to denote elements of C. Moreover, let Δ be a set of non-logical axioms of the form $\forall \overline{x}.(c \supset c')$ where all free variables in c and c' are in $\overline{x}$. We say that d entails *d', written as $d \vdash_\Delta d'$, iff the sequent $\Delta, d \longrightarrow d'$ is probable in LJ [5].*

The language of determinate CCP processes is built from constraints in the underlying constraint system as follows:

$$P, Q ::= \mathbf{tell}(c) \mid \mathbf{ask}\ c\ \mathbf{then}\ P \mid P \parallel Q \mid (\mathbf{local}\, \overline{x})\, P \mid p(\overline{x})$$

where variables in $\overline{x}$ are pairwise distinct. A way to introduce non-determinism in CCP is by adding the usual non-deterministic choice operator $P + P'$. However, as the systems considered here are all determinate, we shall add this operator only when needed.

$$\frac{(X;\Gamma;c) \equiv (X';\Gamma';c') \longrightarrow (Y';\Delta';d') \equiv (Y;\Delta;d)}{(X;\Gamma;c) \to (Y;\Delta;d)}\ \mathrm{R_{EQUIV}}$$

$$\frac{}{(X;\mathbf{tell}(c),\Gamma;d) \longrightarrow (X;\Gamma;c \wedge d)}\ \mathrm{R_T} \qquad \frac{d \vdash_\Delta c}{(X;\mathbf{ask}\ c\ \mathbf{then}\ P,\Gamma;d) \longrightarrow (X;P,\Gamma;d)}\ \mathrm{R_A}$$

$$\frac{x \notin X \cup fv(d) \cup fv(\Gamma)}{(X;(\mathbf{local}\ x)\ P,\Gamma;d) \longrightarrow (X \cup \{x\};P,\Gamma;d)}\ \mathrm{R_L} \qquad \frac{p(\overline{x}) \stackrel{\mathtt{def}}{=} P}{(X;p(\overline{y}),\Gamma;d) \longrightarrow (X;P[\overline{y}/\overline{x}],\Gamma;d)}\ \mathrm{R_C}$$

(a) Operational rules for CCP.

$$\frac{(X;P;c) \longrightarrow (X';P';d)}{(X;[P]_i;c) \longrightarrow (X';[P]_i,P';d)}\ \mathrm{R_E} \qquad \frac{(X;P;d^i) \longrightarrow (X';P';d')}{(X;[P]_i;d) \longrightarrow (X';[P']_i;d \wedge s_i(d'))}\ \mathrm{R_S}$$

(b) Operational rules for `eccp` and `sccp`

$$\frac{}{(X;\Box P;\Gamma;d) \longrightarrow (X;P,\circ\Box P;\Gamma;d)}\ \mathrm{R_\Box} \quad \frac{n \geq 0}{(X;\star P,\Gamma;d) \longrightarrow (X;\circ^n P,\Gamma;d)}\ \mathrm{R_\star} \quad \frac{(\emptyset;P;c) \longrightarrow^* (X;\Gamma;d) \not\longrightarrow}{P \stackrel{(c,\exists X.d)}{\Longrightarrow} (\mathbf{local}\ X)\ F(\Gamma)}\ \mathrm{R_{Obs}}$$

(c) Internal and Observable rules for timed-ccp. $\circ^n$ means $\circ ... \circ$ n-times. $F(\Gamma)$ –the *future* of Γ– is defined as: $F(\mathbf{ask}\ c\ \mathbf{then}\ Q) = \emptyset$, $F(\circ Q) = Q$ and $F(P_1, ..., P_n) = F(P_1) \parallel ... \parallel F(P_n)$

Fig. 1. Operational semantics for CCP calculi

The process $\mathbf{tell}(c)$ adds c to the store d producing the new store $d \wedge c$. The process $\mathbf{ask}\ c\ \mathbf{then}\ P$ evolves into P if the store entails c. Otherwise, it remains blocked until more information is added to the store. This provides a synchronization mechanism based on constraint entailment. The process $(\mathbf{local}\,\overline{x})\ P$ behaves as P and binds the variables in $\overline{x}$ to be local to it. The process $p(\overline{x})$ evolves into $P[\overline{x}/\overline{y}]$ provided the definition $p(\overline{y}) \stackrel{\mathtt{def}}{=} P$ where all free variables of P are in the pairwise distinct variables $\overline{y}$.

The operational semantics of CCP is given by the transition relation $\gamma \longrightarrow \gamma'$ satisfying the rules on Figure 1(a). These rules are straightforward realizing the operational intuitions given above. Moreover, they will form the core of transitions common to the other systems that we encode later. A *configuration* γ is a triple of the form $(X;\Gamma;c)$, where c is a constraint (a logical formula specifying the store), Γ is a multiset of processes, and X is a set of hidden (local) variables of c and Γ. The multiset $\Gamma = P_1, P_2, \ldots, P_n$ represents the process $P_1 \parallel P_2 ... \parallel P_n$. We shall indistinguishably use both notations to denote parallel composition of processes.

Processes are quotiented by a structural congruence relation $\cong$ satisfying: (1) $P \parallel Q \cong Q \parallel P$; (2) $P \parallel (Q \parallel R) \cong (P \parallel Q) \parallel R$; and (3) $(\mathbf{local}\,x)\ P \cong (\mathbf{local}\,y)\ P[y/x]$ if $y \notin fv(P)$. Furthermore, $\Gamma = \{P_1, ..., P_n\} \cong \{P'_1, ..., P'_n\} = \Gamma'$ iff $P_i \cong P'_i$ for all $1 \leq i \leq n$. Finally, $(X;\Gamma;c) \cong (X';\Gamma';c')$ iff $X = X'$, $\Gamma \cong \Gamma'$ and $c \equiv_\Delta c'$ (*i.e.*, $c \vdash_\Delta c'$ and $c' \vdash_\Delta c$).

Let $\longrightarrow^*$ be the reflexive and transitive closure of $\longrightarrow$. If $(X;\Gamma;d) \longrightarrow^* (X';\Gamma';d')$ and $\exists X'.d' \vdash_\Delta c$ we write $(X;\Gamma;d) \Downarrow_c$. If $X = \emptyset$ and $d = 1$ we simply write $\Gamma \Downarrow_c$. Intuitively, if P is a process then $P \Downarrow_c$ captures the outputs of P (under input 1).

4 Encoding CCP Languages as SELL$^{\Cap}$ Formulas

This section gives an interpretation of CCP processes as SELL$^{\Cap}$ formulas. The encoding we propose will be used as basis in the subsequent sections to encode CCP calculi that include modalities. For this, we rely on the two following features of SELL$^{\Cap}$. The first one is the subexponential quantifiers $\Cap$ and $\Cup$, which enable the specification of systems

governing an unbounded number of modalities, *e.g.*, spaces or agents. For instance, these quantifiers allow us to specify that process definitions are available to all entities in the system. The second feature is the presence of non-equivalent subexponential prefixes (such as, *e.g.*, $!^i$ or $!^i?^i$) which will be written generically as $\bigtriangledown_i$. This is the key for encoding correctly the different modalities, such as spatial, epistemic or temporal. As we mentioned before, while in linear logic there are only seven classes of modalities, SELL$^{\Cap}$ (and even *SELL*) allows for an unbounded number of non-equivalent prefixes.

4.1 Basic Encoding

We shall use a signature $\langle I \cup \{nil, \infty\}, \preceq, \{\mathfrak{c}, \mathfrak{p}, \mathfrak{d}\}, U\rangle$ with three families and two distinguished elements *nil* (the least element) and ∞ (the greatest element). Moreover, $\mathfrak{c}(a) \in U$ for all $a \in I \cup \{nil, \infty\}$ and $\mathfrak{p}(\infty) \in U$, while $\mathfrak{p}(nil), \mathfrak{d}(nil) \notin U$. Intuitively, the family $\mathfrak{c}$ is used to mark constraints; the family $\mathfrak{p}$ is used to mark processes; and the family $\mathfrak{d}$ is used to mark procedures $p(\overline{x})$ whose definition $p(\overline{y}) \stackrel{\mathtt{def}}{=} P$ may be unfolded. As it will be clear later, the remaining indices in I specify the modalities available in the system, where *nil* represents no modality. For instance, $\mathfrak{p}(nil)$ will mark a process that is not under any modality. Since process definitions, non-logical axioms and constraints can be used as many times, $\mathfrak{c}(a)$, $\mathfrak{p}(\infty)$ for any $a \in I$ are unbounded; since processes and procedure calls are consumed when executed, $\mathfrak{p}(nil)$ and $\mathfrak{d}(nil)$ are bounded.

Encoding Constraints and Processes Constraints and processes are encoded in SELL$^{\Cap}$ by using two functions: $\mathcal{P}[\![P]\!]_l$ for processes and $C[\![c]\!]_l$ for constraints. These encodings will depend on the system that we want to encode and they are parametric on an index $l \in I$. Next we define such functions for the set of basic processes and constraints shown in Section 3. Later, we refine these encodings by adding new cases handling the specific constraints of each system. These cases will basically play with the index l.

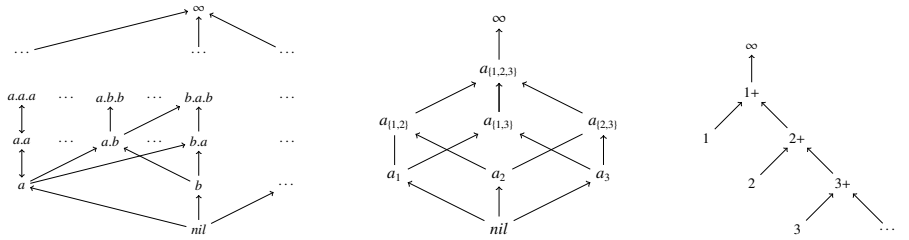
Definition 2 (Encoding of Constraints and Processes). *Let $\langle I \cup \{nil, \infty\}, \preceq, \{\mathfrak{c}, \mathfrak{p}, \mathfrak{d}\}, U\rangle$ be a subexponential signature, and let $l \in I$. For any constraint c, atomic formula A and process P we define $C[\![c]\!]_l$ and $\mathcal{P}[\![P]\!]_l$ as follows:*

$$
\begin{array}{ll}
C[\![c_1 \wedge c_2]\!]_l = C[\![c_1]\!]_l \otimes C[\![c_2]\!]_l & \mathcal{P}[\![\mathbf{tell}(c)]\!]_l = !^{\mathfrak{p}(l)}[\Cap s : l.(C[\![c]\!]_s)] \\
C[\![\exists \overline{x}.c]\!]_l = \exists \overline{x}.C[\![c]\!]_l & \mathcal{P}[\![\mathbf{ask}\ c\ \mathbf{then}\ P]\!]_l = !^{\mathfrak{p}(l)}[\Cap s : l.(C[\![c]\!]_s \multimap \mathcal{P}[\![P]\!]_s)] \\
C[\![A]\!]_l = \bigtriangledown_{\mathfrak{c}(l)} A & \mathcal{P}[\![(\mathbf{local}\ \overline{x})\ P]\!]_l = !^{\mathfrak{p}(l)}[\Cap s : l.\exists \overline{x}.(\mathcal{P}[\![P]\!]_s)] \\
C[\![1]\!]_l = \bigtriangledown_{\mathfrak{c}(l)} 1 & \mathcal{P}[\![P_1, ..., P_n]\!]_l = \mathcal{P}[\![P_1]\!]_l \otimes ... \otimes \mathcal{P}[\![P_n]\!]_l \\
 & \mathcal{P}[\![p(\overline{x})]\!]_l = \bigtriangledown_{\mathfrak{d}(l)} p(\overline{x})
\end{array}
$$

Hence, atomic constraints, processes and procedure calls ($p(\overline{x})$) are marked, respectively, with subexponentials from the $\mathfrak{c}$, $\mathfrak{p}$ and $\mathfrak{d}$ family. The role of the subexponential quantifiers in the encoding will become clear in the following sections. The idea is that they allow choosing in which modality a resulting process should be placed. We note that by using simple logical equivalences, the encoding $C[\![c]\!]_l$ can be rewritten as $\exists \overline{x}.\left[\bigtriangledown_{\mathfrak{c}(l_1)} A_1 \otimes \cdots \otimes \bigtriangledown_{\mathfrak{c}(l_n)} A_n\right]$ where each A_i is an atomic formula or the unit 1.

Encoding Non-Logical Axioms and Process Definitions A non-logical axiom of the form $\forall \overline{x}(d \supset c)$ is encoded as:

$$\Cap l : \infty.\forall \overline{x}.(C[\![c]\!]_l \multimap C[\![d]\!]_l)$$



(a) Epistemic reasoning. (b) Common knowledge. (c) Timed modalities.

Fig. 2. Subexp. signatures for epistemic and time reasoning. Here $a \to b$ denotes that $a \preceq b$.

specifying that the non-logical axiom is available to all subexponentials in the ideal of ∞, *i.e.*, all elements in I. Similarly, a process definition is encoded as:

$$\oiint l : \infty.\forall \overline{x}.(\bigtriangledown_{\mathfrak{d}(l)} p(\overline{x}) \multimap \mathcal{P}[\![P]\!]_l)$$

We write $[\![\Delta]\!]$ and $[\![\Psi]\!]$ for the set of formulas encoding the non-logical axioms Δ and the process definitions Ψ. Finally, a configuration $(X; \Gamma; c)$ is encoded as the sequent:

$$\mathcal{A}; \mathcal{L} \cup X; !^{\mathfrak{c}(\infty)}[\![\Delta]\!], !^{\mathfrak{p}(\infty)}[\![\Psi]\!], \mathcal{P}[\![\Gamma]\!]_{nil}, C[\![c]\!]_{nil} \longrightarrow G$$

The formula G on the right is the goal to be proved, i.e., the encoding of the constraint we are interested to know whether it can be outputted or not by the system. Finally, as normally done [3], the fresh values X are specified as eigenvariables in the logic.

Since the left introduction rules for $\exists$ and $\otimes$ are invertible [1], we can rewrite the sequent above as follows, where we elide the contexts $\mathcal{A}$ and $\mathcal{L} \cup X$:

$$!^{\mathfrak{c}(\infty)}[\![\Delta]\!], !^{\mathfrak{p}(\infty)}[\![\Psi]\!], \mathcal{P}[\![\Gamma]\!]_{nil}, \bigtriangledown_{\mathfrak{c}(l_1)} A_1, \cdots, \bigtriangledown_{\mathfrak{c}(l_n)} A_n \longrightarrow G$$

It is worth noticing that the store is specified by the atomic formulas it contains (A_i), marked with the prefix, $\bigtriangledown_{\mathfrak{c}(l_i)}$. Up to now, from Definition 2, we have a unique l_i, namely *nil*. The forthcoming encodings will enable different subexponential indices to be used, illustrating the encoding's modularity. Moreover, by changing the signature's pre-order, we will be able to specify different modalities (see *e.g.*, Figure 2(a)).

The specification of processes, on the other hand, simply manipulates the set of constraints appearing on the left-hand side of sequents. For instance, the encoding of **tell**(c) adds the atomic constraints which compose c to the left-hand side of the sequent, as in rule R_T. Repeating this process we can prove the following adequacy result.

Theorem 2. *Let P be a CCP process, $(C, \vdash_\Delta)$ be a CS, Ψ be a set of process definitions. Let $\bigtriangledown_l$ be instantiated to $!^l$. Then $P \Downarrow_c$ iff $!^{\mathfrak{c}(\infty)}[\![\Delta]\!], !^{\mathfrak{p}(\infty)}[\![\Psi]\!], \mathcal{P}[\![P]\!]_{nil} \longrightarrow C[\![c]\!]_{nil} \otimes \top$.*

The adequacy that we get is in fact quite strong on the *level of derivations* [11]. It relies on the completeness of the focusing strategy [1] (see the extended version of this paper, available at the authors' personal pages for details). This means that doing proof search from our encoding corresponds exactly to executing processes in the encoded CCP language. This is much stronger than the encoding of CCP in [4], which is only on the *level of provability*. In fact, all our encodings, except the one for `tcc` (Section 4.4), have that strong level of adequacy.

4.2 Epistemic Meaning to Subexponentials

Knight *et al.* in [7] proposed Epistemic CCP (`eccp`), a CCP-based calculi where systems of agents are considered for distributed and epistemic reasoning. In `eccp`, the

constraint system, seen as an Scott information system as in [15], is extended in order to consider spaces of agents. Roughly, each agent i has a space s_i and $s_i(c)$ means "c holds in the space –store– of agent i."

The following definition gives an instantiation of an epistemic constraint system where basic constraints are built as in Definition 1.

Definition 3 (Epistemic Constraint System (ECS)). *Let A be a countable set of agent names. An ECS $(C_e, \vdash_{\Delta_e})$ is a CS where, for any $i \in A$, $s_i : C_e \longrightarrow C_e$ satisfies:*
1. $s_i(1) = 1$ *(bottom preserving)*
2. $s_i(c \wedge d) = s_i(c) \wedge s_i(d)$ *(lub preserving)*
3. *If $d \vdash_{\Delta_e} c$ then $s_i(d) \vdash_{\Delta_e} s_i(c)$ (monotonicity)*
4. $s_i(c) \vdash_{\Delta_e} c$ *(beliefs are facts –extensiveness–)*
5. $s_i(s_i(c)) = s_i(c)$ *(idempotence)*

CCP processes are extended in eccp with the constructor $[P]_i$ that represents P running in the space of the agent i. The operational rules for $[P]_i$ are specified in Figure 1(b). In epistemic systems, agents are trustful, *i.e.*, if an agent i knows some information c, then c is necessarily true. Furthermore, if j knows that i knows c, then j also knows c. For example, given a hierarchy of agents as in $[[P]_i]_j$, it should be possible to propagate the information produced by P in the space i to the outermost space j. This is captured exactly by the rule $\mathrm{R_E}$, which allows a process P in $[P]_i$ to run also outside the space of agent i, i.e., P can be contracted. The rule $\mathrm{R_S}$, on the other hand, allows us to observe the evolution of processes inside the space of an agent. There, the constraint d^i represents the information the agent i may see or have of d, *i.e.*, $d^i = \bigwedge\{c \mid d \vdash_{\Delta_e} s_i(c)\}$. For instance, *$i$ sees c* from the store $s_i(c) \wedge s_j(c')$.

We now configure the encodings in Section 4 so to encode epistemic modalities, starting by the subexponential signature that we use. Let $A = \{a_1, a_2, ...\}$ be a possible infinite set of agents and let A^* be the set of non-empty strings of elements in A; for example, if $a, b \in A$, then $a, b, a.a, b.a, a.b.a, \ldots \in A^*$. We shall use $\bar{i}, \bar{l}, etc$ to denote elements in A^*. We shall also consider *nil* to be the empty string, thus the string $\bar{i}.nil.\bar{l}$ is written as $\bar{i}.\bar{l}$. We let $I = A^* \cup \{nil, \infty\}$ and $U = \{\mathfrak{c}(\bar{l}), \mathfrak{d}(\bar{l}), \mathfrak{p}(\bar{l}) \mid \bar{l} \in I\} \setminus \{\mathfrak{d}(nil), \mathfrak{p}(nil)\}$. Intuitively, the connective $!^{\mathfrak{p}(1.2.3)}$ specifies a process in the structure $[[[\cdot]_3]_2]_1$, denoting "agent 1 knows that agent 2 knows that agent 3 knows" expressions. The connective $!^{\mathfrak{c}(1.2.3)}$, on the other hand, specifies a constraint of the form $s_1(s_2(s_3(\cdot)))$. Notice that all $\mathfrak{p}(\cdot)$ and $\mathfrak{d}(\cdot)$ subexponentials except the ones constructed using *nil* are unbounded. This reflects the fact that both constraints and processes in the space of an agent are unbounded, as specified by rule $\mathrm{R_E}$.

The pre-order $\preceq$ is as depicted in Figure 2(a). More precisely, for every two different agent names a and b in A, the subexponentials a and b are unrelated; Moreover, two sequences in A^* are related $i_1.i_2.\cdots i_m \preceq j_1.j_2.\cdots j_n$ whenever the following sequent is provable $!^{\mathfrak{c}(j_1)}!^{\mathfrak{c}(j_2)}\ldots!^{\mathfrak{c}(j_n)}F \longrightarrow !^{\mathfrak{c}(i_1)}!^{\mathfrak{c}(i_2)}\ldots!^{\mathfrak{c}(i_n)}F$, for any formula F. Alternatively, the pre-order on sequences of agent names could be defined as $a \approx a.\ldots.a$ and $b_1 \ldots b_n \preceq \bar{i}_1.b_1.\bar{i}_2.b_2...\bar{i}_n.b_n$ where each $\bar{i}_i$ is a possible empty string of elements in A.

The shape of the pre-order is key for our encoding. In particular, we are using one subexponential index, *e.g.*, $\mathfrak{p}(i_1.i_2.\cdots i_n)$, to denote a prefix of subexponential bangs $!^{\mathfrak{p}(i_1)}!^{\mathfrak{p}(i_2)}\ldots!^{\mathfrak{p}(i_n)}$. Thus if two subexponentials l, l' are equal in the pre-order ($l \approx l'$), it means that they represent the same equivalence class of prefixes. This way, we are able

to quantify over such prefixes (or boxes) by using a single quantifier, for example, as we do for the encoding of the non-logical axioms and procedure calls.

Definition 4 (Epistemic constraints and processes). *We extend* $C[\![\cdot]\!]_l$ *in Definition 2 so that* $C[\![s_i(c)]\!]_{\overline{l}} = C[\![c]\!]_{\overline{l}.i}$ *and* $\bigtriangledown_{\overline{l}}$ *is instantiated as* $!^{\overline{l}}$*. Moreover, we extend* $\mathcal{P}[\![\cdot]\!]_l$ *in Definition 2 so that* $\mathcal{P}[\![[P]_i]\!]_{\overline{l}} = \mathcal{P}[\![P]\!]_{\overline{l}.i}$.

Observe that, in $\mathcal{P}[\![P]\!]_{\overline{l}}$, $\overline{l}$ is the space-location where P is executed. The role of the quantifier subexponentials in the encoding of processes in Definition 2 is key. For instance, recall that $\mathcal{P}[\![\textbf{ask } c \textbf{ then } P]\!]_l = !^{\mathfrak{p}(l)}[\Cap s : l.(C[\![c]\!]_s \multimap \mathcal{P}[\![P]\!]_s)]$. Here $!^{\mathfrak{p}(l)}$ specifies the epistemic state $[]_l$ where the process is. On the other hand, $\Cap s : l.$, specifies that one can move the process anywhere in the ideal of l. From the pre-order shown in Figure 2(a), this means moving the process to anywhere outside the box $[]_l$. This corresponds exactly to the operational rule $\mathrm{R_E}$. Moreover, since $\mathfrak{p}(l) \in U$, the process is unbounded, thus the encoding $\mathcal{P}[\![\textbf{ask } c \textbf{ then } P]\!]_l$ is not consumed. In fact, the sequent $\mathcal{P}[\![P]\!]_{\overline{l}.i} \longrightarrow \mathcal{P}[\![P]\!]_{\overline{l}}$ is provable for any process P and indexes $\overline{l}$ and i. That is, any process can move to an outer box (see details in the extended version of this paper, available at the authors' personal page).

The following proposition shows that $C[\![\cdot]\!]_{\overline{l}}$, the proposed translation of constraints to formulas in $\mathrm{SELL}^{\Cap}$, represents indeed an epistemic constraint system.

Proposition 1. *Let* $(\mathcal{C}_e, \vdash_{\Delta_e})$ *be an ECS and* $C[\![\cdot]\!]_l$ *be as in Definition 4. Then, for any* $\overline{l}$*:*
1. $C[\![1]\!]_{\overline{l}} \equiv 1$ *(bottom preserving);*
2. $C[\![c \wedge d]\!]_{\overline{l}} \equiv C[\![c]\!]_{\overline{l}} \otimes C[\![d]\!]_{\overline{l}}$ *(lub preserving);*
3. *If* $d \vdash_{\Delta_e} c$ *then* $!^{\mathfrak{c}(\infty)}[\![\Delta_e]\!], C[\![d]\!]_{\overline{l}} \longrightarrow C[\![c]\!]_{\overline{l}}$ *(monotonicity);*
4.$C[\![s_i(c)]\!]_{\overline{l}} \longrightarrow C[\![c]\!]_{nil}$ *(beliefs are facts);*
5. $C[\![s_i(s_i(c))]\!]_{\overline{l}} \equiv C[\![s_i(c)]\!]_{\overline{l}}$ *(idempotence).*

Example 1 (Epistemic Reasoning). Let $P = \textbf{tell}(c)$, $Q = \textbf{ask } c \textbf{ then tell}(d)$ and $R = [P \parallel [Q]_b]_a$. The following sequent is provable $\mathcal{P}[\![R]\!]_{nil} \longrightarrow !^{\mathfrak{c}(a)}c \otimes !^{\mathfrak{c}(nil)}c \otimes \top$. That is, c is *known* by agent a and the external environment (*i.e.*, c is a *fact*). Also, $\mathcal{P}[\![R]\!]_{nil} \longrightarrow !^{\mathfrak{c}(a)}d \otimes \top$ since Q also runs in the scope of a. This intuitively means that a knows that b knows that if c is true, then d is true. Therefore, a knows c and d. Furthermore, the agent b does not know c, *i.e.*, the sequent $\mathcal{P}[\![R]\!]_{nil} \longrightarrow !^{\mathfrak{c}(b)}c \otimes \top$ *is not* provable.

Theorem 3 (Adequacy). *Let* P *be an* `eccp` *process,* $(\mathcal{C}_e, \vdash_e)$ *be an ECS,* Ψ *be a set of process definitions and let* $C[\![\cdot]\!]_{\overline{l}}$ *and* $\mathcal{P}[\![\cdot]\!]_{\overline{l}}$ *be as in Definition 4. Then* $P \Downarrow_c$ *iff* $!^{\mathfrak{c}(\infty)}[\![\Delta_e]\!], !^{\mathfrak{p}(\infty)}[\![\Psi]\!], \mathcal{P}[\![P]\!]_{nil} \longrightarrow C[\![c]\!]_{nil} \otimes \top$.

This result, besides giving an interesting interpretation of subexponentials as knowledge spaces, gives a proof system for the verification of `eccp` processes. Note that, because of the "$\top$" connective, we only consider the observables of a process regardless whether the final configuration has suspended ask processes.

So far, we have assumed that knowledge is not shared by agents. Next example shows how to handle common knowledge among agents. The approach is similar to the one given in [7], by introducing *announcements* of constraints among group of agents, but by using our proof theoretic framework.

Example 2 (Common Knowledge). Assume a finite set of agents $A = \{a_1, ..., a_n\}$ and a process definition: $global_P() \stackrel{\text{def}}{=} P \parallel [P \parallel global_P()]_{a_1} \parallel ... \parallel [P \parallel global_P()]_{a_n}$. For instance, $global_{\mathbf{tell}(c)}$ makes c available in all spaces and nested spaces involving agents in A. Instead of computing common knowledge by recursion, we can complement the subexponential signature as in Figure 2(b) where for all $\mathcal{S} \subseteq A$, $\overline{i} \preceq a_{\mathcal{S}}$ for any string $\overline{i} \in \mathcal{S}^*$. Then, the announcement of c on the group of agents $\mathcal{S}$ can be represented by $!^{\mathfrak{c}(a_{\mathcal{S}})}c$. Notice that the sequent $!^{\mathfrak{c}(a_{\mathcal{S}})}c \longrightarrow !^{\mathfrak{c}(\overline{i})}c$ can be proved for any $\overline{i} \in \mathcal{S}^*$.

4.3 Spaces and Information Confinement

Inconsistent information in CCP arises when considering theories containing axioms such as $c \wedge d \vdash_{\Delta} 0$. Notice that agents are not allowed to tell or ask false, *i.e.*, 0 is not a (basic) constraint. Unlike epistemic scenarios, in spatial computations, a space can be locally inconsistent and it does not imply the inconsistency of the other spaces (*i.e.*, $s_i(0)$ does not imply $s_j(0)$). Moreover, the information produced by a process in a space is not propagated to the outermost spaces. In [7], spatial computations are specified in spatial CCP (sccp) by considering processes of the form $[P]_i$ as in the epistemic case, but excluding the rule $\mathrm{R_E}$ in the system shown in Figure 1(b). Furthermore, some additional requirements are imposed on the representation of agents' spaces ($s_i(\cdot)$).

Definition 5 (Spatial Cons. Sys. (SCS)). *Let A be a countable set of agent names. An SCS $(C_s, \vdash_{\Delta_s})$ is a CS where, for any $i \in A$, $s_i : C_s \longrightarrow C_s$ is a mapping satisfying bottom and lub preserving, monotonicity and* false containment *(see Proposition 2).*

The set $I = A^* \cup \{nil, \infty\}$ is the same as in the encoding of the epistemic case but the pre-order is much simpler: we only require that for any $\overline{i} \in A^*$, $\overline{i} \preceq \infty$. That is, two different elements of A^* are unrelated. Since sccp does not contain the $\mathrm{R_E}$ rule, processes in spaces are treated linearly, i.e., we set $U = \{\mathfrak{c}(l) \mid l \in I\} \cup \{\mathfrak{p}(\infty)\}$. Moreover, the confinement of spatial information is captured by a different subexponential prefix, namely, by instantiating ∇_l as the prefix $!^l?^l$.

Definition 6 (Spatial constraints in SELL$^{\Cap}$). *The encoding $C[\![\cdot]\!]_{\overline{l}}$ maps constraints in a SCS into SELL$^{\Cap}$ formulas and it is defined as in Definition 4. $\mathcal{P}[\![\cdot]\!]_l$ is as in Definition 2 extended with $\mathcal{P}[\![[P]_i]\!]_{\overline{l}} = \mathcal{P}[\![P]\!]_{\overline{l}.i}$. In both cases, however, ∇_l is instantiated as $!^l?^l$.*

Differently from the epistemic case, the encoding of $[P]_i$ runs P only the space of i and not outside it. This is captured by the pre-order above and by instantiating ∇_l as $!^l?^l$. Notice that the ideal of all index l in $I \setminus \{\infty\}$ is the singleton $\{l\}$. This means that the only way of instantiating the subexponential quantifier $(\Cap s : l)$ in the encoding of processes is by using the l itself. In this way, we confine the information inside the location of agents as states the following proposition.

Proposition 2 (False confinement). *Let $(C_s, \vdash_{\Delta_s})$ be a SCS and $C[\![\cdot]\!]_{\overline{l}}$ as in Definition 6. Then, monotonicity, bottom and lub preserving items in Proposition 1 hold. Furthermore, for any $\overline{l} \in A^*$, if we assume that $c \wedge d \vdash_{\Delta_s} 0$:*
1. *$C[\![0]\!]_{\overline{l}} \longrightarrow C[\![c]\!]_{\overline{l}}$ (any c can be deduced in $\overline{l}$ if its local store is inconsistent);*
2. *$C[\![0]\!]_{\overline{l}} \longrightarrow C[\![0]\!]_{\overline{l}}$ is not provable (*false *is confined);*

3. $!^{\mathfrak{c}(\infty)}C[\![\Delta_s]\!], C[\![c]\!]_{\bar{l}}, C[\![d]\!]_{\bar{l}} \longrightarrow C[\![0]\!]_{\bar{l}}$ *($\bar{l}$ becomes inconsistent if it contains c and d);*
4. $!^{\mathfrak{c}(\infty)}C[\![\Delta_s]\!], C[\![c]\!]_{\bar{l}}, C[\![d]\!]_{\bar{l}} \longrightarrow C[\![0]\!]_{\bar{l}}$ *is not provable*
5. $C[\![c]\!]_{\bar{l}} \longrightarrow c$ *and* $C[\![c]\!]_{\bar{l}} \longrightarrow C[\![c]\!]_{nil}$ *are both not provable (local info. is not global).*

Example 3 (Local stores). Let $P = \mathbf{tell}(c)$ and $Q = \mathbf{ask}\ c\ \mathbf{then}\ \mathbf{tell}(d)$. Let $R = [P]_a \parallel [Q]_b$. Then, Q remains blocked since the information c is only available on the space of a. In our encoding, as $!^{\mathfrak{c}(a)}?^{\mathfrak{c}(a)}c \longrightarrow !^{\mathfrak{c}(b)}?^{\mathfrak{c}(b)}c$ is not provable, the sequent $\mathcal{P}[\![R]\!]_{nil} \longrightarrow !^{\mathfrak{c}(b)}?^{\mathfrak{c}(b)}d \otimes \top$ is also not provable. Now let $R = [P]_a \parallel [Q]_a$. The process P adds d in the space of a and then, Q can evolve. Thus, $\mathcal{P}[\![R]\!]_{nil} \longrightarrow !^{\mathfrak{c}(a)}?^{\mathfrak{c}(a)}d \otimes \top$ is provable. Moreover, c does not propagate outside the scope of agent a, *i.e.*, the sequent $\mathcal{P}[\![R]\!]_{nil} \longrightarrow !^{\mathfrak{c}(nil)}?^{\mathfrak{c}(nil)}c \otimes \top$ is not provable. Finally, consider $R = [[P]_a]_b \parallel [Q]_a$. Since $a \not\leq b.a$ and $b.a \not\leq a$, the sequent $!^{\mathfrak{c}(b.a)}?^{\mathfrak{c}(b.a)}c \longrightarrow !^{\mathfrak{c}(a)}?^{\mathfrak{c}(a)}c$ is not provable. Thus, the process Q inside the agent a remains blocked, *i.e.*, the sequent $\mathcal{P}[\![R]\!]_{nil} \longrightarrow !^{\mathfrak{c}(a)}?^{\mathfrak{c}(a)}d \otimes \top$ is not provable. This intuitively means that the space that b confers to a may behave differently (*i.e.*, it contain different information) from the own space of a. The same reasoning applies for the process $R = [[P]_a]_a \parallel [Q]_a$. This means that, in general, the space of a inside a is different from the space a ($a \not\leq a.a$). If we want spaces to be idempotent, we simply need to add the equivalence $a.a \approx a$ to the pre-order.

Theorem 4 (Adequacy). *Let P be an* `sccp` *process, $(C_s, \vdash_s)$ be an SCS, Ψ be a set of process definitions and $C[\![\cdot]\!]_{\bar{l}}$ and $\mathcal{P}[\![\cdot]\!]_{\bar{l}}$ be as in Definition 6. Then $P \Downarrow_c$ iff* $!^{\mathfrak{c}(\infty)}[\![\Delta_s]\!], !^{\mathfrak{p}(\infty)}[\![\Psi]\!], \mathcal{P}[\![P]\!]_{nil} \longrightarrow \mathcal{P}[\![c]\!]_{nil} \otimes \top$.

4.4 Temporal Modalities

Saraswat *et al.* proposed in [17] timed-CCP (`tcc`), an extension of CCP for the specification of reactive systems. In `tcc`, time is conceptually divided into *time intervals* (or *time units*). In a particular time interval, a CCP process P gets an input c from the environment, it executes with this input as the initial *store*, and when it reaches its resting point, it *outputs* the resulting store d to the environment. The resting point determines also a residual process Q which is then executed in the next time unit. The resulting store d is not automatically transferred to the next time unit. Hence, computations during a time-unit proceed monotonically (by adding information to the store), but outputs of two different time-units are not supposed to be related to each other. This view of *reactive computation* is akin to synchronous languages such as Esterel, where the system reacts continuously with the environment at a rate controlled by the environment.

The syntax of CCP is extended in `tcc` by including temporal operators:

$$P, Q ::= \cdots \mid \circ P \mid \Box P$$

The process $\circ P$ delays the execution of P in one time-unit. The replication $\Box P$ means $P \parallel \circ P \parallel \circ \circ P \parallel \ldots$, *i.e.*, unboundly many copies of P, but one at a time.

In `tcc`, recursive calls are assumed to be guarded by a "$\circ$" process to avoid non-terminating sequences of recursive calls during a time-unit. Recursive procedures can then be encoded via replication (see [8]) and we omit them here. We also distinguish between internal ($\longrightarrow$) and observable ($\Longrightarrow$) transitions. The internal transition of the form $(X; \Gamma; c) \longrightarrow (X'; \Gamma'; c')$ is similar to that of CCP plus the additional rules for the timed constructs (see Figure 1(c)). A process $\Box P$ executes one copy of P in the

current time-unit and then, executes again $\Box P$ in the next time-unit (Rule $R_{\Box}$). The seemingly missing rule for $\circ P$ is given by the observable transition relation.

Assume that $(\emptyset; \Gamma; c) \longrightarrow^* (X; \Gamma'; c') \not\longrightarrow$. We say that (the parallel composition in) Γ under input c outputs $\exists X.c'$ and we write $\Gamma \overset{(c,\exists X.c')}{\Longrightarrow} \Upsilon$ where $\Upsilon = (\mathbf{local}\, X)\, F(\Gamma')$ corresponds to the *future* of Γ' (see Figure 1(c)). Roughly, the future function drops any ask whose guard cannot be entailed from the final store. Furthermore, it unfolds the processes guarded by "$\circ$". Note that $F(\cdot)$ does not consider the processes $\mathbf{tell}(c)$, $\Box P$ and $(\mathbf{local}\, x)\, P$ since all of them have an internal transition. Therefore, in a final configuration $(X, \Gamma, c) \not\longrightarrow$ they must occur within the scope of "$\circ$". If, $\Gamma = \Gamma_1 \overset{(1,c_1)}{\Longrightarrow} \Gamma_2 ... \Gamma_n \overset{(1,c_n)}{\Longrightarrow} \Gamma_{n+1}$ and $c_n \vdash_{\Delta} c$, we say that Γ outputs c and we write $\Gamma \Downarrow_c$.

As before, we use a specific subexponential signature but with only two families $\mathfrak{c}$ and $\mathfrak{p}$ as procedure calls are not required as we explained before:

$$I = \{\infty, nil\} \cup \{i, i+ \mid i \geq 1\} \qquad U = \{\mathfrak{c}(i), \mid i \in I\} \cup \{\mathfrak{p}(\infty)\}.$$

Notice that only the subexponentials marking constraints, $\mathfrak{c}(\cdot)$, and boxed processes, $\mathfrak{p}(\infty)$, are unbounded, as they can be used as many times as needed. On the other hand, subexponentials processes, $\mathfrak{p}(\cdot)$, are bounded.

The pre-order is depicted in Figure 2(c), where a descending chain is formed with the numbers marked with "+". Intuitively, the subexponential i is used to specify a given time-unit while $i+$ is used to store processes valid *from* the time-unit i on. This chain captures the semantics of $\Box P$: if $\Box P$ appears in time i, then P should be available at any future time. Formally, that chain allows us to specify, by using a quantifier $\Cap l : i+$, that P can be instantiated anywhere in the ideal of $i+$, *i.e.*, in future time units.

Definition 7 (Timed Constraints in SELL$^{\Cap}$). *We instantiate $\bigtriangledown_l$ as $!^l ?^l$. The interpretation $C[\![\cdot]\!]_l$ is as in Definition 2, while we modify $\mathcal{P}[\![\cdot]\!]_l$ as follows:*

$$\mathcal{P}[\![\mathbf{tell}(c)]\!]_l = !^{\mathfrak{p}(l)} C[\![c]\!]_l \qquad \mathcal{P}[\![\mathbf{ask}\ c\ \mathbf{then}\ P]\!]_l = !^{\mathfrak{p}(l)}(C[\![c]\!]_l \multimap \mathcal{P}[\![P]\!]_l)$$
$$\mathcal{P}[\![(\mathbf{local}\, \overline{x})\, P]\!]_l = !^{\mathfrak{p}(l)}(\exists \overline{x}.(\mathcal{P}[\![P]\!]_l)) \qquad \mathcal{P}[\![\circ P]\!]_i = \mathcal{P}[\![P]\!]_{i+1}$$
$$\mathcal{P}[\![\Box P]\!]_i = !^{\mathfrak{p}(\infty)} \Cap l : i+ (\mathcal{P}[\![P]\!]_l)$$

The encoding of the non-temporal operators are similar as before, just that we do not need the subexponential quantification. While the encoding of $\circ P$ is straightforward, the encoding of $\Box P$ is more interesting. If the process $\Box P$ is executed in the time-unit i, then the encoding of P must be available in subexponentials representing the subsequent time-units. For example, let $P = \Box\, \mathbf{ask}\ c\ \mathbf{then}\ Q$. The process P must execute Q in all time-units $j \geq i$ whenever c can be deduced in j. We make use of universal quantification over locations to capture this behavior.

We note that the observable transition ($\Longrightarrow$) results from a finite sequence of internal ($\longrightarrow$) transitions (R_{Obs} in Figure 1(c)). Proof theoretically, detecting that a given configuration cannot longer be reduced is problematic in general. In fact, the adequacy theorem below is not on the level of derivations, as our previous theorems, but only at the level of provability [11]: P outputs c iff one can prove that there is a time-unit where c holds. Key for proving this theorem is the use of $!^l ?^l$ prefixes as for the `sccp` case. More precisely, facts are confined to a determinate time unit: any formula derived in a subexponential representing a time unit is not spilled to other subexponentials, unless explicitly specified. We note also that we consider here the monotonic fragment of `tcc` *i.e.*, we do not include the time-out **unless** c ($\circ P$) that executes P in the next time-unit

if c cannot be deduced. This operator lacks of a proper proof theoretic semantics: the reduction to P amounts to showing that there is no proof of c.

Theorem 5 (Adequacy). *Let P be a timed process, (C_t, Δ_t) be a CS and $\mathcal{P}[\![\cdot]\!]_l$ as in Definition 7. Then $P \Downarrow_c$ iff $!^{\mathfrak{c}(\infty)}[\![\Delta_t]\!], \mathcal{P}[\![P]\!]_1 \longrightarrow \Cup l : 1+.!^{\mathfrak{c}(l)}?^{\mathfrak{c}(l)}c \otimes \top$.*

5 Concluding Remarks

In this paper, we have introduced quantification over subexponentials in linear logic with subexponentials and proved that cut elimination is admissible for the resulting system (SELL$^{\Cap}$), reflecting a pleasant duality with the standard quantification over terms. We demonstrated that SELL$^{\Cap}$ is, indeed, a powerful tool for specifying concurrent systems involving modalities by proposing novel encodings for CCP-calculi featuring epistemic, spatial and timed modalities, hence providing a proof-theoretic foundation for those calculi.

We believe that there are many directions to follow from this work. For instance, in our encoding, we did not need the generation of *fresh* subexponential variables by using the rules $\Cap_R$ and $\Cup_L$. As done with *eigenvariables* for modeling nonces in security protocol [3], it seems possible to create *new* modalities, such as new spaces or new agents not related to the ones already created as in the Ambient Calculus. This would solve the limitation of `sccp` and `eccp` in [7] where the set of agents is fixed.

Although this paper does not consider non-determinism, some form of it can be easily captured. For instance a non-deterministic choice of the form $P + Q$ can be encoded as the formula $F = \mathcal{P}[\![P]\!]_l \,\&\, \mathcal{P}[\![Q]\!]_l$. In fact, by adding and moving subexponential bangs, it is possible to model precisely don't-care and don't-know choices [10]. Thus, non-determinism (not-considered in [7] for neither `sccp` nor `eccp`) can be also introduced in `sccp`, where processes do not contract. For a second example, consider the `ntcc` calculus [8] which extends `tcc` with guarded non-deterministic choices and asynchrony. For the later, the process $\star P$ represents an arbitrary long, but finite delay for the activation of P; that is, $\star P$ non-deterministically chooses $n \geq 0$ and behaves as $\circ^n P$ (see Rule $\mathrm{R}_\star$ in Figure 1(c)). It seems possible to encode this behavior by extending $\mathcal{P}[\![\cdot]\!]_l$ with the following case: $\mathcal{P}[\![\star P]\!]_i = \Cup l : i+.\mathcal{P}[\![P]\!]_l$. Roughly, if $\star P$ is executed in time-unit i, then *there is* a subexponential j such that $j \preceq i+$ (*i.e.*, a future time-unit j) and the encoding of P holds using that subexponential.

However, for adequacy, some care has to be taken to avoid undesired interactions between $\Box$ and a non-deterministic processes P (containing $\star$ or $+$): $\mathcal{P}[\![\Box P]\!]_l$ yields a formula of the form $!^{\mathfrak{p}(\infty)}F$. Due to the connective $!^{\mathfrak{p}(\infty)}$ that precedes F, by contraction, it is possible to have a derivation with two copies of F representing the process $P \parallel P$ that does not behave as P, thus breaking adequacy.

We think that CCP research can greatly profit from this work. Due to the modularity of our encoding, it seems possible to design variants of CCP by simply configuring the subexponentials differently or by using different prefixes. For instance, by using a mix of linear and unbounded $\mathfrak{c}(\cdot)$ subexponentials, it is possible to specify a spatial CCP language that allows constraints to be consumed. It also seems possible to design CCP models that allow for the creation of new spaces or agents. Finally, one could explore the use of these new variants of CCP and their declarative reading as SELL$^{\Cap}$ formulas to

reason about other models of concurrency (see e.g., [14] that studies different fragments of the asynchronous π-calculus through the CCP model).

Acknowledgments. We thank Sophia Knight, Frank Valencia and Jorge A. Perez for helpful discussions. Nigam was supported by CNPq. Pimentel was supported by FAPEMIG. This work has been (partially) supported by Colciencias (Colombia), and by Digiteo and DGAR (École Polytechnique) funds for visitors.

References

1. Andreoli, J.-M.: Logic programming with focusing proofs in linear logic. J. of Logic and Computation 2(3), 297–347 (1992)
2. Danos, V., Joinet, J.-B., Schellinx, H.: The structure of exponentials: Uncovering the dynamics of linear logic proofs. In: Mundici, D., Gottlob, G., Leitsch, A. (eds.) KGC 1993. LNCS, vol. 713, pp. 159–171. Springer, Heidelberg (1993)
3. Durgin, N.A., Lincoln, P., Mitchell, J.C., Scedrov, A.: Multiset rewriting and the complexity of bounded security protocols. JCS 12(2), 247–311 (2004)
4. Fages, F., Ruet, P., Soliman, S.: Linear concurrent constraint programming: Operational and phase semantics. Information and Computation 165(1), 14–41 (2001)
5. Gentzen, G.: Investigations into logical deductions. In: Szabo, M.E. (ed.) The Collected Papers of Gerhard Gentzen, pp. 68–131. North-Holland, Amsterdam (1969)
6. Girard, J.-Y.: Linear logic. Theoretical Computer Science 50, 1–102 (1987)
7. Knight, S., Palamidessi, C., Panangaden, P., Valencia, F.D.: Spatial and epistemic modalities in constraint-based process calculi. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 317–332. Springer, Heidelberg (2012)
8. Nielsen, M., Palamidessi, C., Valencia, F.D.: Temporal concurrent constraint programming: Denotation, logic and applications. Nordic Journal of Computing 9(1), 145–188 (2002)
9. Nigam, V.: On the complexity of linear authorization logics. In: LICS, pp. 511–520. IEEE (2012)
10. Nigam, V., Miller, D.: Algorithmic specifications in linear logic with subexponentials. In: PPDP, pp. 129–140. ACM (2009)
11. Nigam, V., Miller, D.: A framework for proof systems. J. Autom. Reasoning 45(2), 157–188 (2010)
12. Nigam, V., Pimentel, E., Reis, G.: Specifying proof systems in linear logic with subexponentials. Electr. Notes Theor. Comput. Sci. 269, 109–123 (2011)
13. Olarte, C., Rueda, C., Valencia, F.D.: Models and emerging trends of concurrent constraint programming. Constraints (2013)
14. Palamidessi, C., Saraswat, V.A., Valencia, F.D., Victor, B.: On the expressiveness of linearity vs persistence in the asychronous pi-calculus. In: LICS, pp. 59–68. IEEE Computer Society (2006)
15. Saraswat, V.A., Rinard, M.C., Panangaden, P.: Semantic foundations of concurrent constraint programming. In: POPL, pp. 333–352. ACM (1991)
16. Saraswat, V.A.: Concurrent Constraint Programming. MIT Press (1993)
17. Saraswat, V.A., Jagadeesan, R., Gupta, V.: Timed default concurrent constraint programming. J. Symb. Comput. 22(5/6), 475–520 (1996)
18. Watkins, K., Cervesato, I., Pfenning, F., Walker, D.: A concurrent logical framework I: Judgments and properties. TR CMU-CS-02-101, CMU (2003)

Compositional Choreographies

Fabrizio Montesi[1] and Nobuko Yoshida[2]

[1] IT University of Copenhagen
[2] Imperial College London

Abstract. We propose a new programming model that supports a compositionality of choreographies. The key of our approach is the introduction of partial choreographies, which can mix global descriptions with communications among external peers. We prove that if two choreographies are composable, then the endpoints independently generated from each choreography are also composable, preserving their typability and deadlock-freedom. The usability of our framework is demonstrated by modelling an industrial use case implemented in a tool for Web Services, Jolie.

1 Introduction

Choreography-based programming is a powerful paradigm for designing communicating systems where the flow of communications is defined from a global viewpoint, instead of separately specifying the behaviour of each *endpoint* (peer). The local behaviour of the endpoints can then be automatically generated by means of *EndPoint Projection* (EPP). This paradigm has been used in standards [21,4] and language implementations [11,19,20,8]. Choreographies impact significantly the quality of software: they lower the chance for programming errors and ease their detection [16,6,7].

Previous works provide models for programming implementations of communicating systems with choreographies [6,7]. These models come with a type discipline for checking choreographies against protocol specifications given as session types [12], which are used to verify that the global behaviour of a choreography implements the expected communication flows. For example, a programmer may express a protocol using a *multiparty session type* [13] (or *global type*) such as the following one:

$$\texttt{B -> C}\colon \langle \texttt{string} \rangle; \texttt{C -> B}\colon \langle \texttt{int} \rangle; \texttt{B -> T}\colon \left\{ \begin{array}{l} ok\colon \texttt{B -> T}\colon \langle \texttt{string} \rangle; \texttt{T -> B}\colon \langle \texttt{date} \rangle, \\ quit\colon \texttt{end} \end{array} \right\}$$

Above, `B`, `C` and `T` are *roles* and abstractly represent endpoints in a system. In the protocol, a buyer `B` sends the name of a product to a catalogue `C`, which replies with the price for that product. Then, `B` notifies the transport role `T` of whether the price is accepted or not. In the first case (label ok), `B` sends also a delivery address to `T` and `T` replies with the expected delivery date. Otherwise (label $quit$), the protocol terminates immediately.

To the best of our knowledge, all previous choreography programming models (e.g., [7,6]) require the programmer to implement the behaviour of all roles in a protocol where it is used; e.g., it would not be possible to write the choreography of a system that uses the protocol above but gives the implementations only of roles `C` and `T`, to make

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 425–439, 2013.

those reusable by other programs as software libraries through an API. This seriously hinders the applicability of choreographies in industrial settings, where the interoperability of different systems developed independently is the key. In particular, it is not currently possible to:

- use choreographies to develop software libraries that implement subsets of roles in protocols such that they can be reused from other systems;
- reuse an existing software library that implements subsets of roles in protocols from inside a choreography.

To tackle the issues above, we ask: *Can we design a choreography model in which the EPP of a choreography can be composed with other existing systems?* The main problem is that existing choreography models rely on the complete knowledge of the implementation details of all endpoints to ensure that the systems generated by EPP will behave correctly. This complete knowledge is not available when independently developed implementations of distributed protocols need to be composed. In order to answer our question, we build a model for developing *partial choreographies*. Partial choreographies implement the behaviour of subsets of the roles in the protocols they use. Endpoint implementations are then automatically generated from partial choreographies and composed with other systems, with the guarantee that their overall execution will follow the intended protocols and the behaviour of the originating choreographies.

Main Contributions. We provide the following contributions:

Compositional Choreographies. We introduce a new programming model for choreographies in which the implementation of some roles in protocols can be omitted (§ 3). These *partial choreographies* can then be composed with others through message passing. Our model allows to describe both choreographies with many participants or just a single endpoint. We provide a notion of EPP that produces correct endpoint code from a choreography, and we show that the EPP of a choreography preserves its compositional properties (§ 5). Our model introduces *shared channel mobility* to choreographies, which gains a dynamism when two protocols are composed.

Typing. We provide a type system for checking choreographies against protocol specifications given as multiparty session types [13]. The type system ensures that the composition of different programs implements the intended protocols correctly (§ 4), and that our EPP produces code that follows the behaviour of the originating choreographies. Our framework guarantees that the EPP is still typable (§ 5); therefore, the EPP is reusable as a "black box" composable with other systems and the result of the composition can be checked for errors by referring only to types.

Deadlock-freedom and Progress among Composed Choreographies. In the presence of partial choreographies, we prove that we can (i) capture the existing methodologies for deadlock-freedom in complete choreographies as in [6,7] and (ii) extend the notion of progress for incomplete systems investigated in [13] to choreographies (§ 5). Our results demonstrate for the first time that choreographies can be effectively used also as a tool for progress in a compositional setting, offering a new viewpoint for investigating progress and giving a fresh look to the results in [6,7].

Proofs, auxiliary definitions, and other resources are posted at [1], including an implementation of our use case (§ 2) with Jolie [15,18].

2 Motivations: A Use Case of Compositional Choreographies

We present motivations for this study by reporting a use case from our industry collaborators [14], and informally introducing our model. For clarity, we discuss only its most relevant parts. An extended version can be found at [1].

In our use case a buyer company needs to purchase a product from one of many available seller companies. The use case has two aspects that previous choreography models cannot handle: (i) the system of the buyer company is developed independently from those of the seller companies, and use the latter as software modules without revealing internal implementation details; (ii) depending on the desired product, the buyer company selects a suitable seller company at runtime. We address these issues with *partial choreographies*. A partial choreography implements a subset of the roles in a protocol, leaving the implementation of the other roles to an external system. External systems can be discovered at runtime. In our case, the buyer company will select a seller and then run the protocol from the introduction by implementing only the buyer role B, and rely on the external seller system to implement the other two roles C and T.

Buyer Choreography. We now define a choreography for the buyer company, C_B.

$$
C_B = \begin{array}{lll}
1.\ \mathsf{u}[\mathsf{U}]\ \mathbf{starts}\ \mathsf{pd}[\mathsf{PD}] : a(k);\ \mathsf{u}[\mathsf{U}].prod \texttt{ -> } \mathsf{pd}[\mathsf{PD}].x : k; & & \\
2.\ \mathsf{pd}[\mathsf{PD}]\ \mathbf{starts}\ \mathsf{r}[\mathsf{R}] : b(k');\ \mathsf{pd}[\mathsf{PD}].x \texttt{ -> } \mathsf{r}[\mathsf{R}].y : k';\ \mathsf{r}[\mathsf{R}].\mathsf{find}(y) \texttt{ -> } \mathsf{pd}[\mathsf{PD}].z : k'; & & \\
3.\ \mathsf{pd}[\mathsf{B}]\ \mathbf{req}\ \mathsf{C},\mathsf{T} : z(k'');\quad \mathsf{pd}[\mathsf{B}].x \texttt{ -> } \mathsf{C} : k'';\quad \mathsf{C} \texttt{ -> } \mathsf{pd}[\mathsf{B}].price : k'' & & \\
4.\ \mathsf{if}\ \mathsf{check}(price)@\mathsf{pd}\ \mathsf{then} & & \\
5.\quad \mathsf{pd}[\mathsf{B}] \texttt{ -> } \mathsf{T} : k'' \oplus ok; & \mathsf{pd}[\mathsf{PD}] \texttt{ -> } \mathsf{u}[\mathsf{U}] : k[del]; & \mathsf{pd}[\mathsf{PD}] \texttt{ -> } \mathsf{u}[\mathsf{U}] : k\langle k''[\mathsf{B}]\rangle; \\
6.\quad \mathsf{u}[\mathsf{B}].addr \texttt{ -> } \mathsf{T} : k''; & \mathsf{T} \texttt{ -> } \mathsf{u}[\mathsf{B}].ddate : k'' & \\
7.\ \mathsf{else} & & \\
8.\quad \mathsf{pd}[\mathsf{B}] \texttt{ -> } \mathsf{T} : k'' \oplus quit; & \mathsf{pd}[\mathsf{PD}] \texttt{ -> } \mathsf{u}[\mathsf{U}] : k[quit] &
\end{array}
$$

Above, a purchase in the buyer company is initiated by a user process u. In Line 1, process u and the freshly created process pd (for purchasing department) start a session k by synchronising on shared channel a. Each process is annotated with the role it plays in the protocol that the session implements. Then, still in Line 1, u sends the product $prod$ the user wishes to buy to pd. In Line 2 pd starts a new session k' with a fresh process r (a service registry) through shared channel b. Then, pd forwards the product name to r, which replies with the shared channel of the seller to contact for the purchase.

We refer to statements such as those in Lines 1-2 as complete, since they describe the behaviour of all participants, both sender and receiver(s). On the other hand, the continuation in Lines 3-8 is a partial choreography that relies on the selected external seller to implement the protocol shown in the introduction and perform the purchase.

The partial choreography in Lines 3-8 is depicted as a sequence chart in Fig. 1.a, where dashed lines indicate interactions with external participants. In Line 3 pd *requests* a synchronisation on the shared channel stored in its local variable z to create the new session k'', declaring that it will play role B and that it expects the environment to implement roles C and T. Session k'' proceeds as specified by the protocol in the introduction. First, pd sends the product name stored in x through session k'' to the external process that is playing role C (the product catalogue executed by the seller company). Observe that here we do not specify the process name of the receiver, since that will be established by the external seller system. Then, pd waits to receive the price for the product from the external process playing role C in k''. In Line 4, pd checks whether

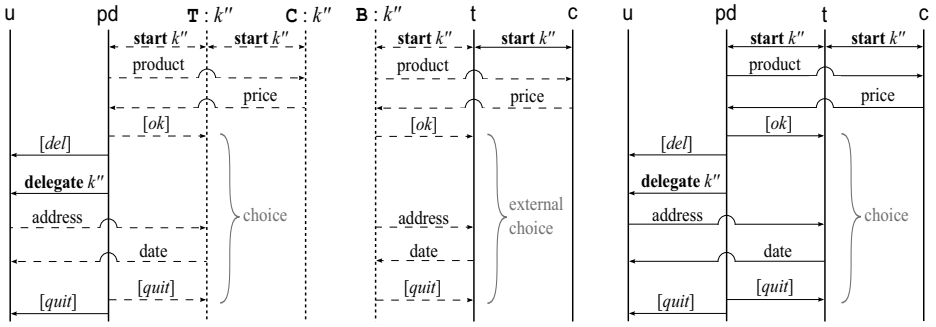


Fig. 1. Sequence charts for buyer (a), seller (b), and their composition (c)

the price is acceptable; if so, in Line 5 pd tells the external process playing role T (the transport process executed by the seller company) and user u (which remains internal to the buyer choreography) to proceed with the purchase (labels ok and del respectively). Still in Line 5, pd *delegates* to u the continuation of session k'' in its place, as role B. In Line 6, the user sends her address to T and receives a delivery date. If the price is not acceptable, Line 7, then in Line 8 pd informs the others to quit the purchase attempt.

Seller Choreography and Composition. We define now a choreography for a seller that can be contacted by C_B. Let the find function in C_B return shared channel c for electronic products, and c' for other products; we refer to the choreographies of the respective seller companies as C_S and C'_S. Below, we define C_S (C'_S, omitted, is similar).

$$C_S = \begin{array}{l} 1.\ \mathbf{acc}\ \mathtt{c[C]}, \mathtt{t[T]} : c(k'');\quad \mathtt{B} \texttt{->} \mathtt{c[C]}.x_2 : k'';\quad \mathtt{c[C]}.\mathsf{price}(x_2) \texttt{->} \mathtt{B} : k'';\\ 2.\ \mathtt{B} \texttt{->} \mathtt{t[T]} : k'' \,\&\, \left\{ \begin{array}{r} ok : \mathtt{B} \texttt{->} \mathtt{t[T]}.daddr : k'';\quad \mathtt{t[T]}.\mathsf{time}(daddr) \texttt{->} \mathtt{B} : k'' \\ quit : \mathbf{0} \end{array} \right\} \end{array}$$

The choreography C_S, depicted as a sequence chart in Fig. 1.b, starts by *accepting* the creation of session k'' through shared channel c, offering to spawn two fresh processes c and t. Choreographies starting with an acceptance act as replicated, modelling typical always-available modules. The acceptance in Line 1 would synchronise with the request made by C_B in the case $z = c$. Afterwards, c expects to receive the product name from the process playing B in session k'', and replies with the respective price. In Line 2, t (the process for the transport) waits for either label ok or $quit$. In the first case, t also waits for a delivery address and then sends back the expected time of arrival.

From the code of C_B and C_S and, graphically, from their respective sequence charts we can see that they are *compatible*: sending actions match receiving actions on the other side and vice versa. Our model can recognise this by using roles in protocols as interfaces between partial choreographies (§ 4). The code for buyer and seller companies can be composed in a network with the parallel operator $\mid$ as: $C = C_B \mid C_S \mid C'_S$. Parallel composition allows partial terms in different choreographies to communicate. In (§ 3, Semantics) we formalise a semantics for choreography composition. To give the intuition behind our semantics, let us consider the sequence charts in Fig. 1.a and Fig. 1.b; their composition will behave as the sequence chart in Fig. 1.c.

$$
\begin{array}{lll|lll}
C ::= \eta; C & & \textit{(seq)} & \mid\ C_1 \mid C_2 & & \textit{(par)} \\
\quad \mid\ \mathsf{if}\ e@\mathtt{p}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 & & \textit{(cond)} & \mid\ (\boldsymbol{\nu} r)\ C & & \textit{(res)} \\
\quad \mid\ \mathsf{rec}\ X(\widetilde{x@\mathtt{p}}, \tilde{k}, \tilde{\mathtt{p}}) = C_2\ \mathsf{in}\ C_1 & & \textit{(rec)} & \mid\ X(\widetilde{x@\mathtt{p}}, \tilde{k}, \tilde{\mathtt{p}}) & & \textit{(call)} \\
\quad \mid\ \mathbf{0} & & \textit{(inact)} & \mid\ \mathtt{A} \texttt{->} q : k\&\{l_i : C_i\}_{i \in I} & & \textit{(branch)} \\
\hline
\eta ::= p\ \mathbf{starts}\ \tilde{q} : a(k) & & \textit{(start)} & \mid\ p.e \texttt{->} q.x : k & & \textit{(com)} \\
\quad \mid\ p \texttt{->} q : k[l] & & \textit{(sel)} & \mid\ p \texttt{->} q : k\langle k'[\mathtt{A}]\rangle & & \textit{(del)} \\
\quad \mid\ p\ \mathbf{req}\ \tilde{\mathtt{B}} : u(k) & & \textit{(req)} & \mid\ \mathbf{acc}\ \tilde{q} : a(k) & & \textit{(acc)} \\
\quad \mid\ p.e \texttt{->} \mathtt{B} : k & & \textit{(com-s)} & \mid\ \mathtt{A} \texttt{->} q.x : k & & \textit{(com-r)} \\
\quad \mid\ p \texttt{->} \mathtt{B} : k\langle k'[\mathtt{C}]\rangle & & \textit{(del-s)} & \mid\ \mathtt{A} \texttt{->} q : k(k'[\mathtt{C}]) & & \textit{(del-r)} \\
\quad \mid\ p \texttt{->} \mathtt{B} : k \oplus l & & \textit{(sel-s)} & & &
\end{array}
$$

$$
p, q ::= \mathtt{p}[\mathtt{A}] \qquad\qquad u ::= x \mid a
$$

Fig. 2. Compositional Choreographies.

3 Compositional Choreographies

This section introduces our model for compositional choreographies, a calculus where complete and partial actions can be freely interleaved.

Syntax. Fig. 2 defines the syntax of our calculus. C is a choreography, η is a complete or partial action, p is a typed process identifier made by a process identifier $\mathtt{p}$ and a role annotation $\mathtt{A}$, k is a session identifier, and a is a shared channel. A term $\eta; C$ denotes a choreography that may execute action η and then proceed as C. In the productions for η, terms *(start)*, *(com)*, *(sel)* and *(del)* are complete actions, whereas all the others are partial. In the productions for C, term *(branch)* is also partial.

Complete Actions. Term *(start)* initiates a session: process $\mathtt{p}$ starts a new multiparty session through shared channel a and tags it with a fresh identifier k. $\mathtt{p}$ is already running and dubbed *active process*, while $\tilde{\mathtt{q}}$ (which we assume nonempty) is a set of bound *service processes* that are freshly created. $\mathtt{A}, \tilde{\mathtt{B}}$ represent the respective roles played by the processes in session k. Term *(com)* denotes a communication where process $\mathtt{p}$ sends, on session k, the evaluation of a first-order expression e to process $\mathtt{q}$, which binds it to its *local variable* x. Expressions may be shared channel names, capturing shared channel mobility. In *(sel)*, $\mathtt{p}$ communicates to $\mathtt{q}$ its selection of branch l. Term *(del)* models session mobility: process $\mathtt{p}$ delegates to $\mathtt{q}$ through session k its role $\mathtt{C}$ in session k'.

Partial Actions. In term *(req)*, process $\mathtt{p}$ is willing to start a new session k by synchronising through shared channel a with some other external processes. $\mathtt{p}$ is willing to play role $\mathtt{A}$ in the session and expects the other processes to play the other roles $\tilde{\mathtt{B}}$. *(req)* terms are supposed to synchronise with always-available *service processes*, modelled by term *(acc)*. In term *(acc)*, processes $\tilde{\mathtt{q}}$ are dynamically spawned whenever requested by a matching *(req)* term on the same shared channel a. Term *(com-s)* models the sending of a message from a process $\mathtt{p}$ to an external process playing role $\mathtt{B}$ in session k. Dually, in *(com-r)* process $\mathtt{q}$ receives a message intended for $\mathtt{B}$ in session k from the external process playing role $\mathtt{A}$. *(del-s)* and *(del-r)* model, respectively, the sending and receiving of a delegation of role $\mathtt{C}$ in session k'. *(sel-s)* models the sending of a selection of label

l. *(sel-s)* can synchronise with a *(branch)* term, which offers a choice on multiple labels. Once a label l_i is selected, *(branch)* proceeds by executing its continuation C_i.

Other Terms. In term *(cond)*, process p evaluates condition e to choose the continuation C_1 or C_2. Term *(res)* restricts the usage of a name r to a choreography C. r can be any name, i.e., a process identifier p, a session identifier k, or a shared channel a. Term *(par)* models the parallel composition of choreographies, allowing partial actions to interact through the network. The other terms are standard: terms *(rec)*, *(call)* and *(inact)* model, respectively, a recursive procedure, a recursive call, and termination.

For clarity, we have annotated process identifiers with roles in all communications. Technically, this is necessary only for terms *(start)*, *(req)* and *(acc)* since roles can be inferred from session identifiers in all other terms (cf. [7]).

Semantics. We give semantics to choreographies with a labelled transition system (lts), whose rules are defined in Fig. 3 and whose labels λ are defined as:

$$\lambda \quad ::= \quad \eta \quad | \quad \mathtt{A} \to q : k\&l \quad | \quad \mathsf{if@p} \quad | \quad (\boldsymbol{\nu} r)\, \lambda$$

We distinguish between labels representing complete or partial actions with the respective sets CAct and PAct. CAct is the smallest set containing all η that are complete actions and the labels of the form $\mathsf{if@p}$, closed under restrictions $(\boldsymbol{\nu} r)$. PAct is the smallest set containing all η that are partial actions and the labels of the form $\mathtt{A} \to q : k\&l$, similarly closed under restriction of names. We also use other auxiliary definitions. $\mathsf{fc}(C)$ returns the set of all session/role pairs $k[\mathtt{A}]$ such that k is free in C and there is a process performing an action as role $\mathtt{A}$ in session k in C. $\mathsf{rc}(\lambda)$ is defined only for partial labels that are not *(req)* or *(acc)*, and returns the session/role pair of the intended external sender or receiver of λ; e.g., $\mathsf{rc}(p.e \to \mathtt{B} : k) = k[\mathtt{B}]$. fn and bn denote the sets of free and bound names in a label or a term. $\mathsf{snd}(\eta)$ returns the name of the sender process in η, and is undefined if η has no sender process (e.g., when η is a *(com-r)*). $\mathsf{rcv}(\eta)$, instead, returns the session/role pair $k[\mathtt{A}]$ where k is the session used in η and $\mathtt{A}$ is the role of the receiver (similarly for $\mathsf{rcv}(\lambda)$). $\mathsf{fc}(\lambda)$ is as $\mathsf{fc}(C)$, but applied on labels.

We comment the rules. Rule $\lfloor^{C}|_{\textsc{Act}}\rceil$ handles actions that can be simply consumed. Rule $\lfloor^{C}|_{\textsc{Start}}\rceil$ starts a session with a global action, by restricting the names of the newly created session identifier k and processes $\tilde{\mathsf{q}}$. Rule $\lfloor^{C}|_{\textsc{Com}}\rceil$ handles the communication of a value by substituting, in the continuation C, the binding occurrence x under process identifier q with value v (evaluated from expression e). Similarly, rules $\lfloor^{C}|_{\textsc{Com-S}}\rceil$ and $\lfloor^{C}|_{\textsc{Com-R}}\rceil$ implement the respective partial sending and receiving actions of a communication. In rule $\lfloor^{C}|_{\textsc{Branch}}\rceil$, process q receives a selection on a branching label and proceeds accordingly. Rules $\lfloor^{C}|_{\textsc{Cond}}\rceil$, $\lfloor^{C}|_{\textsc{Res}}\rceil$, and $\lfloor^{C}|_{\textsc{Ctx}}\rceil$ are standard. Rule $\lfloor^{C}|_{\textsc{Par}}\rceil$ makes global actions observable and blocks partial actions if their counterpart is in the parallel branch C_2. In rule $\lfloor^{C}|_{\textsc{Eq}}\rceil$, the relation $\mathcal{R}$ can either be the swapping relation $\simeq_C$, which swaps terms that describe the behaviour of different processes [7], or the structural congruence $\equiv$, which handles name restriction and recursion unfolding (see [1]).

Rule $\lfloor^{C}|_{\textsc{Sync}}\rceil$ is the main rule and enables two choreographies to perform compatible sending/receiving partial actions λ and λ' to interact and realise a global action, defined by $\lambda \circ \lambda'$. Function $\circ : \mathsf{PAct} \times \mathsf{PAct} \to \mathsf{CAct}$ is formally defined by the rules below:

$$\begin{array}{lcl}
\mathsf{p}[\mathtt{A}] \to \mathtt{B} : k\langle v\rangle \;\circ\; \mathtt{A} \to \mathsf{q}[\mathtt{B}] : k\langle v\rangle & = & \mathsf{p}[\mathtt{A}] \to \mathsf{q}[\mathtt{B}] : k\langle v\rangle \\
\mathsf{p}[\mathtt{A}] \to \mathtt{B} : k\langle k'[\mathtt{C}]\rangle \;\circ\; \mathtt{A} \to \mathsf{q}[\mathtt{B}] : k(k'[\mathtt{C}]) & = & \mathsf{p}[\mathtt{A}] \to \mathsf{q}[\mathtt{B}] : k\langle k'[\mathtt{C}]\rangle \\
\mathsf{p}[\mathtt{A}] \to \mathtt{B} : k \oplus l \;\circ\; \mathtt{A} \to \mathsf{q}[\mathtt{B}] : k\&l & = & \mathsf{p}[\mathtt{A}] \to \mathsf{q}[\mathtt{B}] : k[l]
\end{array}$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Act}}\rceil \quad \eta \notin \{(com),(com\text{-}s),(com\text{-}r),(start),(acc)\} \;\Rightarrow\; \eta; C \xrightarrow{\eta} C$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Start}}\rceil \quad \eta = p\ \mathbf{starts}\ \widetilde{\mathsf{q}[\mathsf{B}]} : a(k) \;\Rightarrow\; \eta; C \xrightarrow{\eta} (\boldsymbol{\nu} k, \tilde{\mathsf{q}})\, C$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Com}}\rceil \quad \eta = p.e \texttt{ -> } \mathsf{q}[\mathsf{B}].x : k \;\Rightarrow\; \eta; C \xrightarrow{p \texttt{ -> } \mathsf{q}[\mathsf{B}]:k\langle v\rangle} C[v/x@\mathsf{q}] \qquad (e \downarrow v)$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Com-S}}\rceil \quad p.e \texttt{ -> } \mathtt{B} : k; C \xrightarrow{p \texttt{ -> } \mathtt{B}:k\langle v\rangle} C \qquad (e \downarrow v)$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Com-R}}\rceil \quad \mathtt{A} \texttt{ -> } \mathsf{q}[\mathsf{B}].x : k; C \xrightarrow{\mathtt{A} \texttt{ -> } \mathsf{q}[\mathsf{B}]:k\langle v\rangle} C[v/x@\mathsf{q}]$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Branch}}\rceil \quad \mathtt{A} \texttt{ -> } q : k\&\{l_i : C_i\}_{i\in I} \xrightarrow{\mathtt{A} \texttt{ -> } q:k\&l_j} C_j \qquad (j \in I)$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Cond}}\rceil \quad \mathsf{if}\ e@\mathsf{p}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \xrightarrow{\mathsf{if}@\mathsf{p}} C_i \qquad (i = 1 \text{ if } e \downarrow \mathrm{true},\ i = 2 \text{ otherwise})$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Res}}\rceil \quad C \xrightarrow{\lambda} C' \;\Rightarrow\; (\boldsymbol{\nu} r)\, C \xrightarrow{(\boldsymbol{\nu} r)\,\lambda} (\boldsymbol{\nu} r)\, C'$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Ctx}}\rceil \quad C_1 \xrightarrow{\lambda} C_1' \;\Rightarrow\; \mathsf{rec}\, X(\widetilde{x@\mathsf{p}}, \tilde{k}, \tilde{\mathsf{p}}) = C_2\ \mathsf{in}\ C_1 \xrightarrow{\lambda} \mathsf{rec}\, X(\widetilde{x@\mathsf{p}}, \tilde{k}, \tilde{\mathsf{p}}) = C_2\ \mathsf{in}\ C_1'$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Par}}\rceil \quad C_1 \xrightarrow{\lambda} C_1' \;\Rightarrow\; C_1 \mid C_2 \xrightarrow{\lambda} C_1' \mid C_2 \qquad (\,\lambda \in \mathsf{CAct} \;\vee\; \mathsf{rc}(\lambda) \notin \mathsf{fc}(C_2)\,)$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Eq}}\rceil \quad \mathcal{R} \in \{\equiv, \simeq_C\} \quad C_1\, \mathcal{R}\, C_1' \quad C_1' \xrightarrow{\lambda} C_2' \quad C_2'\, \mathcal{R}\, C_2 \;\Rightarrow\; C_1 \xrightarrow{\lambda} C_2$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Sync}}\rceil \quad C_1 \xrightarrow{\lambda} C_1' \quad C_2 \xrightarrow{\lambda'} C_2' \;\Rightarrow\; C_1 \mid C_2 \xrightarrow{\lambda\circ\lambda'} C_1' \mid C_2'$$

$$\lfloor^{\mathrm{C}}|_{\textsc{P-Start}}\rceil \quad \left.\begin{array}{l} i \in [1,n] \quad \{\tilde{\mathsf{q}}\} = \{\tilde{\mathsf{q}}_1, \ldots, \tilde{\mathsf{q}}_n\} \\ \{\tilde{\mathsf{B}}\} = \{\tilde{\mathsf{B}}_1, \ldots, \tilde{\mathsf{B}}_n\} \quad C'' = \prod_i C_i \\ C \xrightarrow{p\ \mathbf{req}\ \tilde{\mathsf{B}}:u(k)} C' \\ C_i = \mathbf{acc}\ \widetilde{\mathsf{q}[\mathsf{B}]}_i : a(k); C_i' \end{array}\right\} \Rightarrow \begin{array}{l} C \mid C'' \xrightarrow{\lambda} (\boldsymbol{\nu} k, \tilde{\mathsf{q}})\left(C' \mid \prod_i (C_i')\right) \mid C'' \\ (\,\lambda = p\ \mathbf{starts}\ \widetilde{\mathsf{q}[\mathsf{B}]}_1, \ldots, \widetilde{\mathsf{q}[\mathsf{B}]}_n : a(k)\,) \end{array}$$

$$\lfloor^{\mathrm{C}}|_{\textsc{Async}}\rceil \quad C \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{r})\, C' \;\Rightarrow\; \eta; C \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{r})\, \eta; C' \quad \left(\begin{array}{cc} \mathsf{snd}(\eta) \in \mathsf{fn}(\lambda) & \tilde{r} = \mathsf{bn}(\lambda) \\ \mathsf{rcv}(\eta) \notin \mathsf{fc}(\lambda) & \tilde{r} \notin \mathsf{fn}(\eta) \\ \eta \notin \{(start),(acc)\} & \end{array}\right)$$

Fig. 3. Semantics of Compositional Choreographies

Observe that if $\lambda \circ \lambda'$ is not defined (the actions are incompatible), then the rule cannot be applied. Similarly, $\lfloor^{\mathrm{C}}|_{\textsc{P-Start}}\rceil$ models a session start by synchronising a partial choreography that requests to start a session with other choreographies that can accept the request on the same shared channel. The choreographies accepting the request remain available afterwards, for reuse. Finally, rule $\lfloor^{\mathrm{C}}|_{\textsc{Async}}\rceil$ models asynchrony, allowing the sender process of an interaction η ($\mathsf{snd}(\eta)$) to send a message and then proceed freely before the intended receiver actually receives it. In the rule, we require asynchrony to preserve the message ordering in a session wrt receivers with a causality check ($\mathsf{rcv}(\eta) \notin \mathsf{fc}(\lambda)$).

4 Typing Compositional Choreographies

We now present our typing discipline, which ensures that sessions in a choreography follow protocol specifications given as global types [13,3]. The key advances from

previous work [7] are: (i) introduction of the typing rules for partial choreographies and shared channel passing; and (ii) typing endpoints by local types, which offer transparent compositional properties for the behaviour of each process.

Global and Local Types from [13,3] are defined below:

$$G ::= \mathtt{A} \to \mathtt{B} : \langle U \rangle ; G \mid \mathtt{A} \to \mathtt{B} : \{l_i : G_i\}_{i \in I} \mid \mu\, \mathbf{t}\,.\, G \mid \mathbf{t} \mid \mathsf{end}$$

$$T ::= !\mathtt{A}\langle U \rangle ; T \mid ?\mathtt{A}\langle U \rangle ; T \mid \oplus \mathtt{A}\{l_i : T_i\}_{i \in I} \mid \&\mathtt{A}\{l_i : T_i\}_{i \in I} \mid \mu\, \mathbf{t}\,.\, T \mid \mathbf{t} \mid \mathsf{end}$$

$$S ::= G \mid \mathtt{int} \mid \mathtt{bool} \cdots \qquad U ::= S \mid T$$

G is a global type. $\mathtt{A} \to \mathtt{B} : \langle U \rangle ; G$ abstracts a communication from role $\mathtt{A}$ to role $\mathtt{B}$ with continuation G, where U is the type of the exchanged message. U can either be a sort type S (used for typing values or shared channels), or a local type T (used for typing session delegation). In $\mathtt{A} \to \mathtt{B} : \{l_i : G_i\}_{i \in I}$, role $\mathtt{A}$ selects one label l_i offered by role $\mathtt{B}$ and the global type proceeds as G_i. All other terms are standard.

T denotes a local type. $!\mathtt{A}\langle U \rangle ; T$ represents the sending of a message of type U to role $\mathtt{A}$, with continuation T. Dually, $?\mathtt{A}\langle U \rangle ; T$ represents the receiving of a message of type U from role $\mathtt{A}$. $\oplus \mathtt{A}\{l_i : T_i\}_{i \in I}$ and $\&\mathtt{A}\{l_i : T_i\}_{i \in I}$ abstract the selection and the offering of some branches. The other terms are standard.

To relate a global type to the behaviour of an endpoint, we project a global type G onto a local type that represents the behaviour of a single role. We write $[\![G]\!]_{\mathtt{A}}$ to denote the projection of G onto the role $\mathtt{A}$, which is defined following [10] (cf. [1]).

Type Checking. We now introduce our type checking discipline for checking choreographies against global types. We use two kinds of typing environments, the linear session typing environments Δ and the unrestricted service environments Γ:

$$\Delta ::= \Delta, k[\mathtt{A}] : T \mid \emptyset \qquad \Gamma ::= \Gamma, x@\mathtt{p} : S \mid \Gamma, X : (\Gamma, \Delta) \mid \Gamma, \mathtt{p} : k[\mathtt{A}] \mid \Gamma, a : G\langle \mathtt{A} | \tilde{\mathtt{B}} | \tilde{\mathtt{C}} \rangle \mid \emptyset$$

Δ is standard [3], where $k[\mathtt{A}] : T$ maps a local type T to a role $\mathtt{A}$ in a session k. In Γ, $x@\mathtt{p} : S$ types variable x of process $\mathtt{p}$ with type S. $X : (\Gamma, \Delta)$ types recursive procedure X. $\mathtt{p} : k[\mathtt{A}]$ establishes that process $\mathtt{p}$ *owns* role $\mathtt{A}$ in session k. $a : G\langle \mathtt{A} | \tilde{\mathtt{B}} | \tilde{\mathtt{C}} \rangle$ types a shared channel a with global type G: $\mathtt{A}$ is the role of the active process that starts the session through a; $\tilde{\mathtt{B}}$ are the roles of the service processes; $\tilde{\mathtt{C}}$ are the roles, in $\tilde{\mathtt{B}}$, that a choreography implements for the shared channel a, enabling compositionality of services. Whenever we write $a : G\langle \mathtt{A} | \tilde{\mathtt{B}} | \tilde{\mathtt{C}} \rangle$ in Γ, we assume that $\tilde{\mathtt{C}} \subseteq \tilde{\mathtt{B}}$, $\mathtt{A} \notin \tilde{\mathtt{B}}$, and that $\mathtt{A}, \tilde{\mathtt{B}} = \mathsf{roles}(G)$. $\mathsf{roles}(G)$ returns the set of roles in a global type G.

We can write $\Gamma, \mathtt{p} : k[\mathtt{A}]$ only if $\mathtt{p}$ is not associated to any other role in session k in Γ (a process may only play one role per session). A process $\mathtt{p}$ may however appear more than once in a same Γ, allowing processes to run multiple sessions. As usual, we require all other kinds of occurrences in environments to have disjoint identifiers.

A typing judgement $\Gamma \vdash C \triangleright \Delta$ establishes that a choreography C is well-typed. Intuitively, C is well-typed if shared channels are used according to Γ and sessions are used according to Δ. Δ gives the session types of the free sessions in C. Following the design idea that services should always be available, shared by other models [6,7], we assume that all *(acc)* terms in a choreography are not guarded by other actions. A selection of the rules defining our typing judgement is reported in Fig. 4.

We comment the typing rules. Rule $[{}^{\mathrm{T}}|_{\textsc{Start}}]$ types a *(start)*; $a : G\langle \mathtt{A} | \tilde{\mathtt{B}} | \tilde{\mathtt{B}} \rangle$ checks that the choreography should implement all roles in protocol G; processes $\tilde{\mathtt{q}}$ are checked to

$$\lfloor^{\mathrm{T}}|_{\textsc{Start}}\rceil\ \frac{\Gamma, a : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle, \Gamma' \vdash C \triangleright \Delta, \Delta' \quad \mathsf{r}[\mathtt{C}] \in \mathsf{p}[\mathtt{A}], \widetilde{\mathsf{q}[\mathtt{B}]} \Leftrightarrow \left(\mathsf{r} : k[\mathtt{C}] \in \Gamma' \wedge k[\mathtt{C}] : [\![G]\!]_{\mathtt{C}} \in \Delta'\right) \quad \tilde{\mathsf{q}} \notin \Gamma}{\Gamma, a : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{B}}\rangle \vdash \mathsf{p}[\mathtt{A}]\ \mathbf{starts}\ \widetilde{\mathsf{q}[\mathtt{B}]} : a(k); C \triangleright \Delta}$$

$$\lfloor^{\mathrm{T}}|_{\textsc{Sel}}\rceil\ \frac{j \in I \quad \Gamma \vdash \mathsf{p} : k[\mathtt{A}], \mathsf{q} : k[\mathtt{B}] \quad \Gamma \vdash C \triangleright \Delta, k[\mathtt{A}] : T_j, k[\mathtt{B}] : T'_j}{\Gamma \vdash \mathsf{p}[\mathtt{A}] \texttt{ -> } \mathsf{q}[\mathtt{B}] : k[l_j]; C \triangleright \Delta, k[\mathtt{A}] : \oplus \mathtt{B}\{l_i : T_i\}_{i \in I}, k[\mathtt{B}] : \&\mathtt{A}\{l_i : T'_i\}_{i \in I}}$$

$$\lfloor^{\mathrm{T}}|_{\textsc{Req}}\rceil\ \frac{\Gamma \vdash x@\mathsf{p} : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\emptyset\rangle \quad \Gamma, \mathsf{p} : k[\mathtt{A}] \vdash C \triangleright \Delta, k[\mathtt{A}] : [\![G]\!]_{\mathtt{A}}}{\Gamma \vdash \mathsf{p}[\mathtt{A}]\ \mathbf{req}\ \tilde{\mathtt{B}} : x(k); C \triangleright \Delta} \qquad \lfloor^{\mathrm{T}}|_{\textsc{Par}}\rceil\ \frac{\Gamma, \Gamma_i \vdash C_i \triangleright \Delta_i}{\Gamma, \Gamma_1 \circ \Gamma_2 \vdash C_1 \mid C_2 \triangleright \Delta_1, \Delta_2}$$

$$\lfloor^{\mathrm{T}}|_{\textsc{Acc}}\rceil\ \frac{\Gamma, a : G\langle \mathtt{D}|\tilde{\mathtt{B}}|\emptyset\rangle, \Gamma' \vdash C \triangleright \Delta, \Delta' \quad \mathsf{r}[\mathtt{C}] \in \widetilde{\mathsf{q}[\mathtt{A}]} \Leftrightarrow \left(\mathsf{r} : k[\mathtt{C}] \in \Gamma' \wedge k[\mathtt{C}] : [\![G]\!]_{\mathtt{C}} \in \Delta'\right) \quad \tilde{\mathsf{q}} \notin \Gamma}{\Gamma, a : G\langle \mathtt{D}|\tilde{\mathtt{B}}|\tilde{\mathtt{A}}\rangle \vdash \mathbf{acc}\ \widetilde{\mathsf{q}[\mathtt{A}]} : a(k); C \triangleright \Delta}$$

$$\lfloor^{\mathrm{T}}|_{\textsc{Com-s}}\rceil\ \frac{\Gamma \vdash e@\mathsf{p} : S \quad \Gamma \vdash \mathsf{p} : k[\mathtt{A}] \quad \Gamma \vdash C \triangleright \Delta, k[\mathtt{A}] : T \quad \mathsf{q} : k[\mathtt{B}] \notin \Gamma}{\Gamma \vdash \mathsf{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B} : k; C \triangleright \Delta, k[\mathtt{A}] : !\mathtt{B}\langle S\rangle; T} \qquad \lfloor^{\mathrm{T}}|_{\textsc{Zero}}\rceil\ \frac{\mathsf{cosha}(\Gamma) \quad \Delta\ \mathsf{end\ only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta}$$

$$\lfloor^{\mathrm{T}}|_{\textsc{Branch}}\rceil\ \frac{i \in I \quad \Gamma \vdash C_i \triangleright \Delta, k[\mathtt{A}] : T_i \quad I \subseteq J \quad \mathsf{p} : k[\mathtt{A}] \notin \Gamma}{\Gamma \vdash \mathtt{A} \texttt{ -> } \mathsf{q}[\mathtt{B}] : k\&\{l_i : C_i\}_{i \in I} \triangleright \Delta, k[\mathtt{B}] : \&\mathtt{B}\{l_j : T_j\}_{j \in J}}$$

Fig. 4. Typing Rules for Compositional Choreographies (selection)

be fresh ($\tilde{\mathsf{q}} \notin \Gamma$); the continuation C is checked by updating Γ' and Δ' respectively with the process ownerships for their roles in k and the local types for their behaviour in k. $\lfloor^{\mathrm{T}}|_{\textsc{Sel}}\rceil$ deals with selection, checking that the selected label l_j is specified in the local types. In rule $\lfloor^{\mathrm{T}}|_{\textsc{Req}}\rceil$, we check that the choreography requesting the services is not responsible for implementing them, to avoid deadlocks due to the lack of services in parallel required by rule $\lfloor^{\mathrm{C}}|_{\textsc{p-Start}}\rceil$, and that the requesting process behaves as expected by its role in the protocol. Conversely, $\lfloor^{\mathrm{T}}|_{\textsc{Acc}}\rceil$ types an *(acc)* term by ensuring that all the roles for which the choreography is responsible are implemented (the other checks are similar to $\lfloor^{\mathrm{T}}|_{\textsc{Start}}\rceil$). This *distribution* of the responsibilities for implementing the different roles in a protocol is handled by rule $\lfloor^{\mathrm{T}}|_{\textsc{Par}}\rceil$, using the role distribution function $\Gamma_1 \circ \Gamma_2$. Formally, $\Gamma_1 \circ \Gamma_2$ is defined as the union of Γ_1 and Γ_2 except for the typing of shared channels with the same name, which are merged with the following rule:

$$a : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle = a : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{D}}\rangle \circ a : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{E}}\rangle \qquad (\tilde{\mathtt{C}} = \tilde{\mathtt{D}} \uplus \tilde{\mathtt{E}})$$

In rule $\lfloor^{\mathrm{T}}|_{\textsc{Zero}}\rceil$ we check that all responsibilities have been implemented and that the sessions in Δ have been executed. Specifically, predicate $\mathsf{cosha}(\Gamma)$ checks that for every $a : G\langle \mathtt{A}|\tilde{\mathtt{B}}|\tilde{\mathtt{C}}\rangle$ in Γ either (i) $\tilde{\mathtt{C}} = \tilde{\mathtt{B}}$, meaning that a was used only internally with *(start)* terms; or (ii) $\tilde{\mathtt{C}} = \emptyset$, meaning that a is used compositionally in collaboration with other choreographies and all roles that the current choreography is responsible for ($\tilde{\mathtt{C}}$) have been implemented correctly with *(acc)* terms. Rules $\lfloor^{\mathrm{T}}|_{\textsc{Com-s}}\rceil$ and $\lfloor^{\mathrm{T}}|_{\textsc{Branch}}\rceil$ type respectively a sending action and a branching. They are very similar to their complete versions since local types allow us to look at the behaviour of processes independently. They also check that the counterpart for the partial action is not in the continuation, by ensuring that there is not process q such that q plays the other role for session k in Γ, which could obviously lead to a deadlock because process p would not have another process to communicate with in parallel as required by rule $\lfloor^{\mathrm{C}}|_{\textsc{Sync}}\rceil$.

Typing Expressiveness. Our typing system exploits the global information given by complete terms and seamlessly falls back to typical session typing when dealing with partial actions. In particular, $\lfloor {}^{\mathrm{T}}|_{\textsc{Sel}} \rceil$ judges that a choice in a protocol is implemented correctly even if only one of the branches is actually followed. This is sound because we are typing a complete term, and therefore we know that the other branches are not used. This expressiveness is typical of choreography-based models [6,7]. However, such a global knowledge is not available in a partial choreography. For example, in rule $\lfloor {}^{\mathrm{T}}|_{\textsc{Branch}} \rceil$ we cannot know which branch will be selected by the sender and we must therefore require that the receiver process supports at least all the branches specified by the corresponding local type, as in standard session typing for endpoints [12,13].

Properties. We conclude this section by presenting the expected main properties of our type system. Below, to state session fidelity, we use the transition of local types $\Delta \xrightarrow{\alpha} \Delta'$ (defined as [13] and fully given in [1]), where α types a partial or complete action. $\alpha \vdash \lambda$ judges that the label λ is for the same session as α and respects its roles and carried type. We also extend our typing judgement with the extra environment Σ, for handling session ownerships with asynchronous delegations at runtime (see [1]).

Theorem 1 (Typing Soundness). *Let $\Gamma; \Sigma \vdash C \triangleright \Delta$. Then,*

- (Subject Swap) *$C \simeq_C C'$ implies $\Gamma; \Sigma \vdash C' \triangleright \Delta$.*
- *$C \xrightarrow{\lambda} C'$ implies that there exists Δ' such that*
  - (Subject Reduction) *$\Gamma'; \Sigma' \vdash C' \triangleright \Delta'$ for some Γ', Σ';*
  - (Session Fidelity) *if λ is a communication on session k, then $\Delta \xrightarrow{\alpha} \Delta'$ with $\alpha \vdash \lambda$; else, $\Delta = \Delta'$.*

5 Properties of Compositional Choreographies

This section states the main properties of our framework wrt the execution of actual systems composed by endpoints.

Endpoint Projection (EPP) generates correct endpoint code from a choreography. Formally, by endpoint code we refer to choreographies that do not contain complete actions. To define the complete EPP, we first define how the behaviour of a single process in a choreography can be projected. We denote this *process projection* of a process p in a choreography C with $[\![C]\!]_{\mathsf{p}}$. Selected rules of process projection are given below:

$$[\![\mathtt{p[A]}\ \mathbf{starts}\ \widetilde{\mathtt{q[B]}} : a(k); C]\!]_{\mathsf{r}} = \begin{cases} \mathtt{p[A]}\ \mathbf{req}\ \tilde{\mathtt{B}} : a(k); [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathbf{acc}\ \mathsf{r}[\mathtt{C}] : a(k); [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r}[\mathtt{C}] \in \widetilde{\mathtt{q[B]}} \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$[\![\mathtt{p[A]}.e\ \texttt{->}\ \mathtt{q[B]}.x : k; C]\!]_{\mathsf{r}} = \begin{cases} \mathtt{p[A]}.e\ \texttt{->}\ \mathtt{B} : k; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathtt{A}\ \texttt{->}\ \mathtt{q[B]}.x : k; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{q} \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$[\![\mathtt{p[A]}.e\ \texttt{->}\ \mathtt{B} : k; C]\!]_{\mathsf{r}} = \begin{cases} \mathtt{p[A]}.e\ \texttt{->}\ \mathtt{B} : k; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$[\![\mathtt{A}\ \texttt{->}\ \mathtt{q[B]}.x : k; C]\!]_{\mathsf{r}} = \begin{cases} \mathtt{A}\ \texttt{->}\ \mathtt{q[B]}.x : k; [\![C]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ [\![C]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{if}\ e@\mathsf{p}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2]\!]_{\mathsf{r}} = \begin{cases} \mathsf{if}\ e@\mathsf{p}\ \mathsf{then}\ [\![C_1]\!]_{\mathsf{r}}\ \mathsf{else}\ [\![C_2]\!]_{\mathsf{r}} & \text{if } \mathsf{r} = \mathsf{p} \\ [\![C_1]\!]_{\mathsf{r}} \sqcup [\![C_2]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

$$[\![\mathtt{A}\ \texttt{->}\ \mathtt{q[B]} : k\&\{l_i : C_i\}_{i \in I}]\!]_{\mathsf{r}} = \begin{cases} \mathtt{A}\ \texttt{->}\ \mathtt{q[B]} : k\&\{l_i : [\![C_i]\!]_{\mathsf{r}}\}_{i \in I} & \text{if } \mathsf{r} = \mathsf{q} \\ \bigsqcup_{i \in I} [\![C_i]\!]_{\mathsf{r}} & \text{otherwise} \end{cases}$$

Process projection follows the structure of the originating choreography. In a *(start)*, we project the active process $\mathtt{p}$ to a request and the service processes $\tilde{\mathtt{q}}$ to (always-available) accepts. In a *(com)*, the sender is projected to a partial sending action and the receiver to a partial receiving action. The projections of *(sel)* and *(del)*, omitted, follow the same principle. Above we also report the rule for projecting *(com-s)* and *(com-r)* to exemplify how we treat partial choreographies: these are simply projected as they are for their respective process, following the structure of the choreography. The projections of conditionals and partial branchings are the only special cases. In a conditional, we project it as it is for the process evaluating the condition, but for all other we merge their behaviours with the *merging* partial operator $\sqcup$ [6]. $C \sqcup C'$ is defined only for partial choreographies that define the behaviour of a single process and returns a choreography isomorphic to C and C' up to branching, where all branches with distinct labels are also included. We use $\sqcup$ also in the projection of *(branch)* terms, where we require the behaviour of all processes not receiving the selection to be merged. As an example, the process projection for process $\mathtt{u}$ in the choreography C_B from our example in § 2 is:

$$[\![C_B]\!]_{\mathtt{u}} = \begin{array}{l} \mathtt{u}[\mathtt{U}]\ \mathbf{req}\ \mathtt{PD} : a(k);\ \mathtt{u}[\mathtt{U}].prod\ \texttt{->}\ \mathtt{PD} : k; \\ \mathtt{PD}\ \texttt{->}\ \mathtt{u}[\mathtt{U}] : k\& \left\{ \begin{array}{l} del : \mathtt{PD}\ \texttt{->}\ \mathtt{u}[\mathtt{U}] : k\langle k''[\mathtt{B}]\rangle;\ \mathtt{u}[\mathtt{B}].addr\ \texttt{->}\ \mathtt{T} : k''; \\ \qquad \mathtt{T}\ \texttt{->}\ \mathtt{u}[\mathtt{U}].ddate : k'', \\ quit : \mathbf{0} \end{array} \right\} \end{array}$$

Using process projection, we can now define the EPP of a whole system. Since different service processes may be started through *(start)* terms on the same shared channel and play the same role, we use $\sqcup$ for merging their behaviours into a single service. We identify these processes with the service grouping operator $\lfloor C \rfloor^a_{\mathtt{A}}$, which computes the set of all service process names in a start or a request in C on shared channel a playing role $\mathtt{A}$. Formally, EPP is the endofunction $[\![C]\!]$ defined in the following.

Definition 1 (Endpoint Projection). *Let $C \equiv (\boldsymbol{\nu}\tilde{a}, \tilde{k}, \tilde{\mathtt{p}})\ C_f$, where C_f does not contain* (res) *terms. Then, the EPP of C is:*

$$[\![C]\!] = (\boldsymbol{\nu}\tilde{a}) \left((\boldsymbol{\nu}\tilde{k}, \tilde{\mathtt{p}}) \left(\prod_{\mathtt{p} \in \mathsf{fn}(C_f)} [\![C_f]\!]_{\mathtt{p}} \right) \mid \prod_{a,\mathtt{A}} \left(\bigsqcup_{\mathtt{p} \in \lfloor C_f \rfloor^a_{\mathtt{A}}} [\![C_f]\!]_{\mathtt{p}} \right) \right)$$

The EPP of a choreography C is the parallel composition of (i) the projections of all active processes and (ii) the merged projections of all service processes started under same shared channel and role. EPP respects the following Lemma, which shows that our model can adequately capture not only typical complete choreographies, but also scale down to describing the behaviour of a single endpoint.

Lemma 1 (Endpoint Choreographies). *Let C be restriction-free, contain only partial terms, and be well-typed. If one of the following two conditions apply, then $C = [\![C]\!]$.*

1. *$C = \mathbf{acc}\ \mathtt{q}[\mathtt{B}] : a(k); C'$ and $\mathtt{q}$ is the only free process name in C';*
2. *otherwise, C has only one free process name.*

We refer to choreographies that respect one of the two conditions above as *endpoint choreographies*. They implement either the behaviour of a single always-available service process (1), or that of a single free process (2). The EPP for these choreographies is the identity since they already model the behaviour of only one endpoint.

The projection of services may lead to undesirable behaviour if service roles for shared channels are not distributed correctly. For example, if we put the choreography C_B from § 2 in parallel with a choreography with a conflicting service on shared channel b for role R (which is internally implemented in C_B) we obtain a race condition, *even if protocols are correctly implemented*. Consider the following choreography:

$$C_R = \mathbf{acc}\ \mathsf{h}[\mathtt{R}] : b(k');\ \mathtt{PD} \texttt{->} \mathsf{h}[\mathtt{R}].x : k';\ \mathsf{h}[\mathtt{R}].c \texttt{->} \mathtt{PD} : k'$$

If we put the projection of C_B in parallel with that of C_R, we get a race condition between the service processes r and h for role R on shared channel b. Hence, the projection of process pd may synchronise with the service offered by C_R for creating session k', instead of that by the projection of service process r in C_B. Consequently, C_B may not follow its intended behaviour. The distribution of service roles performed by our type system avoids this kind of situations. Observe that normal session typing cannot help us in detecting these problems, because the service process h correctly implements the same communication behaviour for session k' as service process r.

Main Theorems. We can now present our main theorems. We build our results on the foundation that the EPP of a choreography is still typable. As in previous work [16,6,7], we need to consider that in the projection of complete choreographies, due to merging, some projected processes may still offer branches that the original complete choreography has discarded with a conditional. Therefore, we state our type preservation result below under the *minimal typing* of choreographies $\vdash_{\mathsf{min}}$, in which the branches in rules $\lfloor {}^{\mathrm{T}} |_{\textsc{Sel}} \rceil$ and $\lfloor {}^{\mathrm{T}} |_{\textsc{Branch}} \rceil$ are typed using the respective minimal branch types.

Theorem 2 (EPP Type Preservation). *Let $\Gamma \vdash_{\mathsf{min}} C \triangleright \Delta$. Then, $\Gamma \vdash_{\mathsf{min}} [\![C]\!] \triangleright \Delta$.*

By Theorem 2, it follows that Theorem 1 applies also to the EPP of a choreography. We use this result to prove that EPP correctly implements the behaviour of the originating choreography, by establishing a formal relation between their respective semantics.

Theorem 3 (EPP Theorem). *Let $C \equiv (\boldsymbol{\nu} \tilde{a}, \tilde{k}, \tilde{\mathsf{p}})\ C_f$, where C_f is restriction-free, be well-typed. Then,*

1. *(Completeness) $C \xrightarrow{\lambda} C'$ implies $[\![C]\!] \xrightarrow{\lambda} \succ [\![C']\!]$.*
2. *(Soundness) $[\![C]\!] \xrightarrow{\lambda} C'$ implies $C \xrightarrow{\lambda} C''$ and $[\![C'']\!] \prec C'$.*

Above, the *pruning relation* $C \prec C'$ is a strong typed bisimilarity [6] such that C has some unused branches and always-available accepts. $\succ$ is a shortcut for $\prec$ interpreted in the opposite direction.

Deadlock-freedom and Progress. We introduce our results on deadlock-freedom and progress mentioned in the Introduction. First, we define deadlock-freedom:

Definition 2 (Deadlock-freedom). *We say that choreography C is deadlock-free if either (i) $C \equiv \mathbf{0}$ or (ii) there exist C' and λ such that $C \xrightarrow{\lambda} C'$ and C' is deadlock-free.*

In our semantics (Fig. 3) complete terms can always be executed; therefore, choreographies that do not contain partial terms, or *complete choreographies*, are deadlock-free:

Theorem 4 (Deadlock-freedom for Complete Choreographies). *Let C be a complete choreography and contain no free variable names. Then, C is deadlock-free.*

By Theorems 3 and 4 we can obtain, as a corollary, that the EPP of well-typed complete choreographies never deadlock.

Corollary 1 (Deadlock-freedom for EPP). *Let C be a complete choreography, contain no free variable names, and be well-typed. Then, $[\![C]\!]$ is deadlock-free.*

Our model can also be used to talk of deadlock-freedom *compositionally*. In a compositional setting, a choreography may get stuck because of partial actions that need to be executed in parallel composition with other choreographies. We say that a choreography can *progress* if it can be composed with another choreography such that (i) all free names can be restricted and the resulting system is still well-typed, ensuring that protocols are implemented correctly; and (ii) the composition is deadlock-free. Differently from deadlock-freedom for complete choreographies, progress for partial choreographies does not follow directly from the semantics. For example, the following choreography does not have the progress property:

$$\mathtt{A} \texttt{ -> } \mathtt{q}[\mathtt{B}] : k;\ \mathtt{p}[\mathtt{A}].e \texttt{ -> } \mathtt{B} : k$$

Above, $\mathtt{q}$ is waiting for a message on session k from $\mathtt{A}$, but that role is implemented by process $\mathtt{p}$ in the continuation. Thus, the two partial actions will never synchronise. As shown in § 4, our type system takes care of checking that roles in sessions or services are distributed correctly, avoiding cases such as this one and ensuring progress. In general, if a well-typed choreographies does not contain inner *(par)* terms we know that it can progress, since role distribution ensures that there exists a compatible environment.

Theorem 5 (Progress for Partial Choreographies). *Let C be a choreography, be well-typed, and contain no* (par) *terms. Then, there exists C' such that $(\boldsymbol{\nu}\tilde{r})\ (C \mid C')$ with $\tilde{r} = \mathsf{fn}(C \mid C')$, is well-typed and deadlock-free.*

By Theorems 2 and 5, it follows as a corollary that also the EPP of a well-typed choreography can progress:

Corollary 2 (Progress for EPP). *Let C contain no free variable names, be well-typed, and contain no* (par) *terms. Then, there exists C' such that $(\boldsymbol{\nu}\tilde{r})\ ([\![C]\!] \mid C')$ with $\tilde{r} = \mathsf{fn}(C \mid C')$, is well-typed and deadlock-free.*

Correctness of Choreography Composition. We end this section by presenting results that allow to reason about the composition of choreographies.

Lemma 2 (Compositional EPP). *Let $C = C_1 \mid C_2$ be well-typed. Then, $[\![C]\!] \equiv [\![C_1]\!] \mid [\![C_2]\!]$.*

By combining Lemma 2 with the Theorems shown so far, we get the following corollary, which summarises the properties for well-typed compositions of choreographies.

Corollary 3 (Compositional Choreographies). *Let $C \mid C'$ be well-typed. Then,*

1. *(EPP Type Preservation) $[\![C]\!] \mid [\![C']\!]$ is well-typed.*
2. *(Completeness) $C \mid C' \xrightarrow{\lambda} C''$ implies $[\![C]\!] \mid [\![C']\!] \xrightarrow{\lambda}\succ [\![C'']\!]$.*
3. *(Soundness) $[\![C]\!] \mid [\![C']\!] \xrightarrow{\lambda} C''$ implies $C \xrightarrow{\lambda} C'''$ and $[\![C''']\!] \prec C''$.*

Our corollary above formally addresses the issues mentioned in the Introduction. Choreographies (C and C' in the corollary) can be developed independently and then their respective projections can be composed.

6 Related Work

Previous works have tackled the problem of defining a formal model for choreographies and giving a correct EPP [7,6]. The main difference wrt our work is compositionality: previous models can only capture closed systems, and do not treat a methodology for composing choreographies. A major difficulty wrt composition given by the approach in [7] is that the EPP of a choreography could be untypable with known type systems for session types. Typability of EPP is important to achieve composition, since a programmer may need to reuse a choreography *after* it has been projected. [7] is the only previous work providing an asynchronous semantics for multiparty sessions in choreographies; however, asynchrony is modelled in two different ways in the choreography model and the endpoint model, raising complexity. As a consequence, the EPP Theorem in [7] has a more complex formulation with weak transitions and confluence, whereas ours can be formulated in a stronger form where EPP mimics its original choreography step by step. [6] preserves typability of projections but does not handle neither asynchrony nor multiparty sessions; instead, they type choreographies with binary sessions. We have shown that choreographies can be made compositional by introducing partial terms to perform message passing with the environment, and that it is possible to ensure typability of EPP in a multiparty and asynchronous setting. This is the first work introducing a compositional multiparty session typing for choreographies, exploiting the projection of global types onto local types. Finally, neither of [7,6] handles shared channel passing, and does not treat how to handle delegation in a compositional setting, where sessions may be delegated to external or internal processes.

Multiparty session types have been previously used for typing endpoint programs [13,3,9]. In our setting, endpoint programs can be captured as special cases of partial choreographies. Our global types are taken from [3]. Differently from our framework, these works capture asynchronous communications with dedicated processes that model order-preserving message queues. An approach more similar to ours can be found in the notion of *delayed input* presented in [17]. [3] defines a type system for progress by building additional restrictions on top of standard multiparty session typing; our model yields a simpler analysis, since we can rely on the fact that complete terms in a choreography do not get stuck. Nevertheless, [3] can capture sessions started by more than one active thread. We leave an extension of our model in this direction as future work.

In [2] the authors use a concept similar to our partial choreographies for protocol specifications, to allow a single process to implement more than one role in a protocol. Differently from our approach, these are not fully-fledged system implementations but abstract behavioural types, which are then used to type check endpoint code. In our setting, the techniques in [2] can be seen as a more flexible way of handling the projection from global types to local types. An extension of our type system to allow for a process to play more than one role in a session as in [2,9] is an interesting future work.

The relationship between choreographies and endpoints has been explored in, among others, [5,16,13,6,7]. Our work distinguishes itself by adopting the same calculus for describing choreographies and endpoints, simplifying the technical development.

Acknowledgements. Yoshida has been partially supported by the Ocean Observatories Initiative and EPSRC EP/K011715/1, EP/K034413/1 and EP/G015635/1.

References

1. Additional Resources, http://www.itu.dk/people/fabr/papers/compchor/
2. Baltazar, P., Caires, L., Vasconcelos, V.T., Vieira, H.T.: A Type System for Flexible Role Assignment in Multiparty Communicating Systems. In: Proc. of TGC (2012)
3. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008), Long version at http://www.di.unito.it/~dezani/papers/cdy12.pdf
4. Business Process Model and Notation, http://www.omg.org/spec/BPMN/2.0/
5. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and orchestration conformance for system design. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 63–81. Springer, Heidelberg (2006)
6. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centered programming for web services. ACM Trans. Program. Lang. Syst. 34(2), 8 (2012)
7. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: POPL, pp. 263–274 (2013)
8. Chor. Programming Language, http://www.chor-lang.org/
9. Deniélou, P.-M., Yoshida, N.: Dynamic multirole session types. In: Proc. of POPL, pp. 435–446. ACM (2011)
10. Deniélou, P.-M., Yoshida, N., Bejleri, A., Hu, R.: Parameterised multiparty session types. LMCS 8(4) (2012)
11. Honda, K., Mukhamedov, A., Brown, G., Chen, T.-C., Yoshida, N.: Scribbling interactions with a formal foundation. In: Natarajan, R., Ojo, A. (eds.) ICDCIT 2011. LNCS, vol. 6536, pp. 55–75. Springer, Heidelberg (2011)
12. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type disciplines for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
13. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proc. of POPL, vol. 43(1), pp. 273–284. ACM (2008)
14. italianaSoftware. http://www.italianasoftware.com/.
15. Jolie. Java Orchestration Language Interpreter Engine, http://www.jolie-lang.org/
16. Lanese, I., Guidi, C., Montesi, F., Zavattaro, G.: Bridging the gap between interaction- and process-oriented choreographies. In: Proc. of SEFM, pp. 323–332. IEEE (2008)
17. Merro, M., Sangiorgi, D.: On asynchrony in name-passing calculi. Mathematical Structures in Computer Science 14(5), 715–767 (2004)
18. Montesi, F., Guidi, C., Zavattaro, G.: Composing Services with JOLIE. In: Proc. of ECOWS, pp. 13–22 (2007)
19. PI4SOA (2008), http://www.pi4soa.org
20. Savara. JBoss Community, http://www.jboss.org/savara/
21. W3C WS-CDL Working Group. Web services choreography description language version 1.0 (2004), http://www.w3.org/TR/ws-cdl-10/

On Negotiation as Concurrency Primitive

Javier Esparza[1] and Jörg Desel[2]

[1] Fakultät für Informatik, Technische Universität München, Germany
[2] Fakultät für Mathematik und Informatik, FernUniversität in Hagen, Germany

Abstract. We introduce negotiations, a model of concurrency close to Petri nets, with multiparty negotiation as primitive. We study the problems of soundness of negotiations and of, given a negotiation with possibly many steps, computing a *summary*, i.e., an equivalent one-step negotiation. We provide a complete set of reduction rules for sound, acyclic, weakly deterministic negotiations and show that, for deterministic negotiations, the rules compute the summary in polynomial time.

1 Introduction

Many modern distributed systems consist of components whose behavior is only partially known. Typical examples include open systems where programs (e.g. Java applets) can enter or leave, multi-agent systems, business processes, or protocols for conducting elections and auctions. An interaction between a fixed set of components with not fully known behavior can be abstractly described as a *negotiation* in which several *parties* (the components involved in the negotiation) nondeterministically agree on an *outcome*, which results in a transformation of internal states of the parties. A more technical but less suggestive term would be a *synchronized nondeterministic choice* and, as the name suggests, these interactions can be modelled in any standard process algebra as a combination of parallel composition and nondeterministic choice, or as small Petri nets. We argue that much can be gained by studying formal models with *negotiation atoms* as concurrency primitive. In particular, we show that the negotiation point of view reveals new classes of systems with polynomial analysis algorithms.

Negotiation atoms can be combined into *distributed negotiations*. For instance, a distributed negotiation between a buyer, a seller, and a broker, consists of one or more rounds of atoms involving the buyer and the broker or the seller and the broker, followed by a final atom between the buyer and the seller. We introduce a formal model for distributed negotiations, close to a colored version of van der Aalst's *workflow nets* [1], and investigate two important analysis problems. First, just like workflow nets, distributed negotiations can be *unsound* because of deadlocks or livelocks (states from which no deadlock is reached, but the negotiation cannot be completed). The *soundness* problem consists of deciding if a given negotiation is sound. Second, a sound negotiation is equivalent to a negotiation with only one atom whose state transformation function determines the possible final internal states of all parties as a function of their initial internal states. We call this negotiation a *summary*. The *summarization problem* consists

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 440–454, 2013.

of computing a summary of a distributed negotiation. Both problems will shown to be PSPACE-hard for arbitrary negotiations, and NP-hard for acyclic ones. They can be solved by means of well-known algorithms based on the exhaustive exploration of the state space. However, this approach badly suffers from the state-explosion problem: even the analysis of distributed negotiations with a very simple structure requires exponential time.

In this paper we suggest *reduction* algorithms that avoid the construction of the state space but exhaustively apply syntactic reduction rules that simplify the system while preserving some aspects of the behavior, like absence of deadlocks. This approach has been extensively applied to Petri nets or workflow nets, but most of this work has been devoted to the liveness or soundness problems [5,15,16,13,22]. For these problems many reduction rules are known, and some sets of rules have been proved *complete* for certain classes of systems [14,10,11], meaning that they reduce all live or sound systems in the class, and only those, to a trivial system (in our case to a single atomic negotiation). However, many of these rules, like the linear dependency rule of [11], cannot be applied to the summarization problem, because they preserve only the soundness property.

We present a complete set of reduction rules for the summarization problem of *acyclic* negotiations that are either *deterministic* or *weakly deterministic*. The rules are inspired by reduction rules used to transform finite automata into regular expressions by eliminating states [18]. In deterministic negotiations all involved agents are deterministic, meaning that they are never ready to engage in more than one atomic negotiation. Intuitively, nondeterministic agents may be ready to engage in multiple atomic negotiations, and which one takes place is decided by the deterministic parties, which play thus the role of negotiation leaders. In weakly deterministic negotiations not every agent is deterministic, but some deterministic party is involved in every atomic negotiation an agent can engage in next. For *deterministic* negotiations we prove that a sound and acyclic negotiation can be summarized by means of a polynomial number of application of the rules, leading to a polynomial algorithm.

The paper is organized as follows. Section 2 introduces the syntax and semantics of the model. Section 3 introduces the soundness and summarization problems. Section 4 presents our reduction rules. Section 5 defines (weakly) deterministic negotiations. Section 6 proves the completeness and polynomial complexity results announced above. Finally, Section 7 presents some conclusions, open questions and related work. The paper only contains proof sketches. Full proofs can be found in [12].

2 Negotiations: Syntax and Semantics

We fix a finite set A of *agents* representing potential parties of negotiations. Each agent $a \in A$ has a (possibly infinite) nonempty set Q_a of *internal states*. We denote by Q_A the cartesian product $\prod_{a \in A} Q_a$. A *transformer* is a left-total relation $\tau \subseteq Q_A \times Q_A$, representing a nondeterministic state transforming function. Given $S \subseteq A$, we say that a transformer τ is an *S-transformer* if, for each

$a_i \notin S$, $\Big((q_{a_1}, \ldots, q_{a_i}, \ldots, q_{a_{|A|}}), (q'_{a_1}, \ldots, q'_{a_i}, \ldots, q'_{a_{|A|}})\Big) \in \tau$ implies $q_{a_i} = q'_{a_i}$. So an S-transformer only transforms the internal states of agents in S.

Definition 1. *A* negotiation atom*, or just an* atom*, is a triple $n = (P_n, R_n, \delta_n)$, where $P_n \subseteq A$ is a nonempty set of* parties*, R_n is a finite, nonempty set of* outcomes*, and δ_n is a mapping assigning to each outcome r in R_n a P_n-transformer $\delta_n(r)$. We denote the transformer $\delta_n(r)$ by $\langle n, r\rangle$, and, if there is no confusion, by $\langle r\rangle$.*

Intuitively, if the states of the agents before a negotiation n are given by a tuple q and the outcome of the negotiation is r, then the agents change their states to q' for some $(q, q') \in \langle n, r\rangle$. Only the parties of n can change their internal states. Each outcome $r \in R_n$ is possible, independent from the previous internal states of the parties.

For a simple example, consider a negotiation atom $n_{\texttt{FD}}$ with parties `F` (Father) and `D` (teenage Daughter). The goal of the negotiation is to determine whether `D` can go to a party, and the time at which she must return home. The possible outcomes are $\{\texttt{yes}, \texttt{no}, \texttt{ask\_mother}\}$. Both sets $Q_{\texttt{F}}$ and $Q_{\texttt{D}}$ contain a state *angry* plus a state t for every time $T_1 \leq t \leq T_2$ in a given interval $[T_1, T_2]$. Initially, `F` is in state t_f and `D` in state t_d. The transformer $\delta_{n_{\texttt{FD}}}$ is given by

$$\begin{aligned}\langle \texttt{yes}\rangle &= \{((t_f, t_d), (t, t)) \mid t_f \leq t \leq t_d \vee t_d \leq t \leq t_f\}\\ \langle \texttt{no}\rangle &= \{((t_f, t_d), (\mathit{angry}, \mathit{angry}))\}\\ \langle \texttt{ask\_mother}\rangle &= \{((t_f, t_d), (t_f, t_d))\}\end{aligned}$$

That is, if the outcome is `yes`, then `F` and `D` agree on a time t which is not earlier and not later than both suggested times. If it is `no`, then there is a quarrel and both parties get angry. If the outcome is `ask_mother`, then the parties keep their previous times.

2.1 Combining Atomic Negotiations

If the outcome of the atom above is `ask_mother`, then $n_{\texttt{FD}}$ should be followed by a second atom $n_{\texttt{DM}}$ between `D` and `M` (Mother). The complete negotiation is the composition of $n_{\texttt{FD}}$ and $n_{\texttt{DM}}$, where the possible internal states of `M` are the same as those of `F` and `D`, and $n_{\texttt{DM}}$ is a copy of $n_{\texttt{FD}}$, but without the `ask_mother` outcome. In order to compose atoms, we add a *transition function* $\mathcal{X}$ that assigns to every triple (n, a, r) consisting of an atom n, a participant a of n, and an outcome r of n a set $\mathcal{X}(n, a, r)$ of atoms. Intuitively, this is the set of atomic negotiations agent a is ready to engage in after the atom n, if the outcome of n is r.

Definition 2. *Given a finite set of atoms N, let $T(N)$ denote the set of triples (n, a, r) such that $n \in N$, $a \in P_n$, and $r \in R_n$. A* negotiation *is a tuple $\mathcal{N} = (N, n_0, n_f, \mathcal{X})$, where $n_0, n_f \in N$ are the* initial *and* final *atoms, and $\mathcal{X}\colon T(N) \to 2^N$ is the* transition function*. Further, $\mathcal{N}$ satisfies the following properties:*

(1) every agent of A participates in both n_0 and n_f;

(2) for every $(n, a, r) \in T(N)$: $\mathcal{X}(n, a, r) = \emptyset$ iff $n = n_f$.

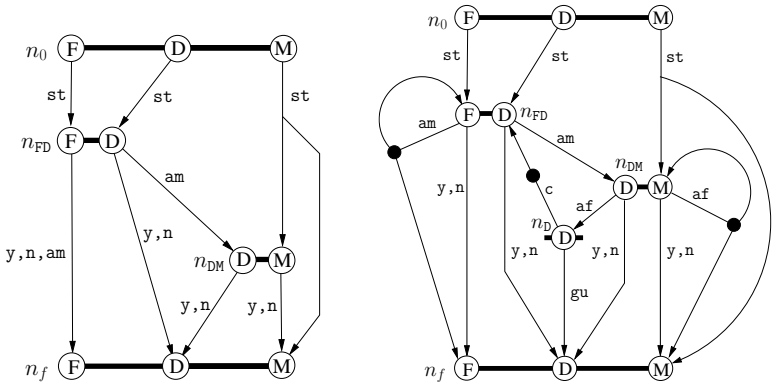


Fig. 1. An acyclic negotiation and the ping-pong negotiation

We may have $n_0 = n_f$. Notice that n_f has, as all other atoms, at least one outcome $r \in R_{n_f}$.

Negotiations are graphically represented as shown in Figure 1. For each atom $n \in N$ we draw a black bar; for each party a of P_n we draw a white circle on the bar, called a *port*. For each $(n, a, r) \in T(N)$, we draw a hyperarc leading from the port of a in n to all the ports of a in the atoms of $\mathcal{X}(n, a, r)$, and label it by r. Figure 1 shows on the left the graphical representation of the Father-Daughter-Mother negotiation sketched above. Instead of multiple (hyper)arcs connecting the same input port to the same output ports we draw a single (hyper)arc with multiple labels. In the figure, we write `y` for `yes`, `n` for `no`, and `am` for `ask_mother`. `st` stands for `start`, the only outcome of n_0. Since n_f has no outgoing arc, the outcomes of n_f do not appear in the graphical representation.

Definition 3. *The* graph associated to a negotiation $\mathcal{N} = (N, n_0, n_f, \mathcal{X})$ *is the directed graph with vertices* N *and edges* $\{(n, n') \in N \times N \mid \exists\, (n, a, r) \in T(N)\colon n' \in \mathcal{X}(n, a, r)\}$. *The negotiation* $\mathcal{N}$ *is* acyclic *if its graph has no cycles.*

The negotiation on the left of Figure 1 is acyclic. The negotiation on the right (ignore the black dots on the arcs for the moment) is the ping-pong negotiation, well-known in every family. The $n_{\mathtt{DM}}$ atom has now an extra outcome `ask_father` (`af`), and Daughter `D` can be sent back and forth between Mother and Father. After each round, `D` "negotiates with herself" (atom $n_{\mathtt{D}}$) with possible outcomes `c` (`continue`) and `gu` (`give up`). This negotiation is cyclic because, for instance, we have $\mathcal{X}(n_{\mathtt{FD}}, \mathtt{D}, \mathtt{am}) = \{n_{\mathtt{DM}}\}$, $\mathcal{X}(n_{\mathtt{DM}}, \mathtt{D}, \mathtt{af}) = \{n_{\mathtt{D}}\}$, and $\mathcal{X}(n_{\mathtt{D}}, \mathtt{D}, \mathtt{c}) = \{n_{\mathtt{FD}}\}$.

2.2 Semantics

A *marking* of a negotiation $\mathcal{N} = (N, n_0, n_f, \mathcal{X})$ is a mapping $\boldsymbol{x}\colon A \to 2^N$. Intuitively, $\boldsymbol{x}(a)$ is the set of atoms that agent a is currently ready to engage in next. The *initial* and *final* markings, denoted by $\boldsymbol{x}_0$ and $\boldsymbol{x}_f$ respectively, are given by $\boldsymbol{x}_0(a) = \{n_0\}$ and $\boldsymbol{x}_f(a) = \emptyset$ for every $a \in A$.

A marking $\boldsymbol{x}$ *enables* an atom n if $n \in \boldsymbol{x}(a)$ for every $a \in P_n$, i.e., if every agent that parties in n is currently ready to engage in it. If $\boldsymbol{x}$ enables n, then n can take place and its parties agree on an outcome r; we say that (n, r) *occurs*. The occurrence of (n, r) produces a next marking $\boldsymbol{x}'$ given by $\boldsymbol{x}'(a) = \mathcal{X}(n, a, r)$ for every $a \in P_n$, and $\boldsymbol{x}'(a) = \boldsymbol{x}(a)$ for every $a \in A \setminus P_n$. We write $\boldsymbol{x} \xrightarrow{(n,r)} \boldsymbol{x}'$ to denote this, and call it a *small step*.

By this definition, $\boldsymbol{x}(a)$ is always either $\{n_0\}$ or equals $\mathcal{X}(n, a, r)$ for some atom n and outcome r. The marking $\boldsymbol{x}_f$ can only be reached by the occurrence of (n_f, r) (r being a possible outcome of n_f), and it does not enable any atom. Any other marking that does not enable any atom is considered a *deadlock*.

Reachable markings can be graphically represented by placing tokens (black dots) on the forking points of the hyperarcs (or in the middle of an arc). Thus, both the initial marking and the final marking are represented by no tokens, and all other reachable markings are represented by exactly one token per agent.

Figure 1 shows on the right the marking in which Father (F) is ready to engage in the atoms $n_{\mathtt{FD}}$ and n_f, Daughter (D) is only ready to engage in $n_{\mathtt{FD}}$, and Mother (M) is ready to engage in both $n_{\mathtt{DM}}$ and n_f.

We write $\boldsymbol{x}_1 \xrightarrow{\sigma}$ to denote that there is a sequence

$$\boldsymbol{x}_1 \xrightarrow{(n_1,r_1)} \boldsymbol{x}_2 \xrightarrow{(n_2,r_2)} \cdots \xrightarrow{(n_{k-1},r_{k-1})} \boldsymbol{x}_k \xrightarrow{(n_k,r_k)} \boldsymbol{x}_{k+1} \cdots$$

of small steps such that $\sigma = (n_1, r_1) \ldots (n_k, r_k) \ldots$. If $\boldsymbol{x}_1 \xrightarrow{\sigma}$, then σ is an *occurrence sequence* from the marking $\boldsymbol{x}_1$, and $\boldsymbol{x}_1$ enables σ. If σ is finite, then we write $\boldsymbol{x}_1 \xrightarrow{\sigma} \boldsymbol{x}_{k+1}$ and say that $\boldsymbol{x}_{k+1}$ is *reachable* from $\boldsymbol{x}_1$. If $\boldsymbol{x}_1$ is the initial marking then we call σ *initial occurrence sequence*. If moreover $\boldsymbol{x}_{k+1}$ is the final marking, then σ is a *large step*.

3 Analysis Problems

Correct negotiations should be deadlock-free and, in principle, they should not have infinite occurrence sequences either. However, requiring the latter in our negotiation model is too strong, because infinite occurrence sequences may be excluded by fairness constraints. Following [1,2], we introduce a notion of partial correctness independent of termination:

Definition 4. *A negotiation is* sound *if* (a) *every atom is enabled at some reachable marking, and* (b) *every occurrence sequence from the initial marking is either a large step or can be extended to a large step.*

The negotiations of Figure 1 are sound. However, if we set $\mathcal{X}(n_0, \mathtt{M}, \mathtt{st}) = \{n_{\mathtt{DM}}\}$ instead of $\mathcal{X}(n_0, \mathtt{M}, \mathtt{st}) = \{n_{\mathtt{DM}}, n_f\}$, then the occurrence sequence $(n_0, \mathtt{st})(n_{\mathtt{FD}}, \mathtt{yes})$ leads to a deadlock.

The *final outcomes* of a negotiation are the outcomes of its final atom. Intuitively, two sound negotiations over the same agents are equivalent if they have the same final outcomes, and for each final outcome they transform the same initial states into the same final states.

Definition 5. *Given a negotiation $\mathcal{N} = (N, n_0, n_f, \mathcal{X})$, we attach to each outcome r of n_f a* summary transformer *$\langle \mathcal{N}, r \rangle$ as follows. Let E_r be the set of large steps of $\mathcal{N}$ that end with (n_f, r). We define $\langle \mathcal{N}, r \rangle = \bigcup_{\sigma \in E_r} \langle \sigma \rangle$, where for $\sigma = (n_1, r_1) \dots (n_k, r_k)$ we define $\langle \sigma \rangle = \langle n_1, r_1 \rangle \cdots \langle n_k, r_k \rangle$ (each $\langle n_i, r_i \rangle$ is a relation on Q_A; concatenation is the usual concatenation of relations).*

$\langle \mathcal{N}, r \rangle(q_0)$ is the set of possible final states of the agents after the negotiation concludes with outcome r, if their initial states are given by q_0.

Definition 6. *Two negotiations $\mathcal{N}_1$ and $\mathcal{N}_2$ over the same set of agents are* equivalent*, denoted by $\mathcal{N}_1 \equiv \mathcal{N}_2$, if they are either both unsound, or if they are both sound, have the same final outcomes, and $\langle \mathcal{N}_1, r \rangle = \langle \mathcal{N}_2, r \rangle$ for every final outcome r. If $\mathcal{N}_1 \equiv \mathcal{N}_2$ and $\mathcal{N}_2$ consists of a single atom, then $\mathcal{N}_2$ is a* summary *of $\mathcal{N}_1$.*

Notice that, according to this definition, all unsound negotiations are equivalent. This amounts to considering soundness essential for a negotiation: if it fails, we do not care about the rest.

3.1 Deciding Soundness

The *reachability graph* of a negotiation $\mathcal{N}$ has all markings reachable from $\boldsymbol{x}_0$ as vertices, and an arc leading from $\boldsymbol{x}$ to $\boldsymbol{x}'$ whenever $\boldsymbol{x} \xrightarrow{(n,r)} \boldsymbol{x}'$.

The soundness problem consists of deciding if a given negotiation is sound. It can be solved by (1) computing the reachability graph of $\mathcal{N}$ and (2a) checking that every atom appears at some arc, and (2b) that, for every reachable marking $\boldsymbol{x}$, there is an occurrence sequence σ such that $\boldsymbol{x} \xrightarrow{\sigma} \boldsymbol{x}_f$.

Step (1) needs exponential time, and steps (2a) and (2b) are polynomial in the size of the reachability graph. So the algorithm is single exponential in the number of atoms. This cannot be easily avoided, because the problem is PSPACE-complete, and NP-complete for acyclic negotiations.

Recall that a language L is in the class DP if there exist languages L_1, L_2 in NP and co-NP, respectively, such that $L = L_1 \cap L_2$ [20].

Theorem 1. *The soundness problem is PSPACE-complete. For acyclic negotiations, the problem is co-NP-hard and in DP (and so at level Δ_2^P of the polynomial hierarchy).*

Proof. (Sketch) Membership in PSPACE follows easily from Savitch's theorem and closure of NPSPACE under complement. For PSPACE-hardness, given a deterministic linearly bounded automaton A and a word w, we construct a negotiation $\mathcal{N}_{A,w}$ such that A accepts w iff $\mathcal{N}_{A,w}$ is sound. $\mathcal{N}_{A,w}$ has an agent C (for control), an agent H (for head), and an agent T_i for every tape cell of A. The negotiation has an atom $n[q, h, a]$ for every state q, head position h, and input letter a, plus initial and final atoms. The transition function is defined so that $\mathcal{N}_{A,w}$ simulates the computation of A on w.

Membership in DP follows easily from the fact that checking the first (second) condition in the definition of soundness lies in NP (co-NP, respectively). Co-NP-hardness follows by a standard reduction from 3-CNF-UNSAT. □

3.2 A Summarization Algorithm

The *summarization problem* consists of computing a summary of a given negotiation, if it is sound. A straightforward solution follows these steps:

(1) Compute the reachability graph of $\mathcal{N}$. Interpret it as a weighted finite automaton over the alphabet of transformers $\langle n, r\rangle$, with $\boldsymbol{x}_0$ as initial state, and $\boldsymbol{x}_f$ as final state.
(2) Compute the sum over all paths σ leading from $\boldsymbol{x}_0$ to $\boldsymbol{x}_f$ of the transformers $\langle\sigma\rangle$. We recall a well-known algorithm for this based on state elimination (see e.g. [18]). The algorithm proceeds in phases consisting of the following three steps:
   (2.1) Exhaustively replace steps $\boldsymbol{x} \xrightarrow{f_1} \boldsymbol{x}'$, $\boldsymbol{x} \xrightarrow{f_2} \boldsymbol{x}'$ by one step $\boldsymbol{x} \xrightarrow{f_1+f_2} \boldsymbol{x}'$.
   (2.2) Pick a state $\boldsymbol{x}$ different from $\boldsymbol{x}_0$ and $\boldsymbol{x}_f$. If there is a self-loop $\boldsymbol{x} \xrightarrow{f} \boldsymbol{x}$, replace all steps $\boldsymbol{x} \xrightarrow{g} \boldsymbol{x}'$, where $\boldsymbol{x}' \neq \boldsymbol{x}$, by the step $\boldsymbol{x} \xrightarrow{g^*f} \boldsymbol{x}'$, and then remove the self-loop.
   (2.3) For every two steps $\boldsymbol{x}_1 \xrightarrow{f_1} \boldsymbol{x}$ and $\boldsymbol{x} \xrightarrow{f_2} \boldsymbol{x}_2$, add a step $\boldsymbol{x} \xrightarrow{f_1 f_2} \boldsymbol{x}_2$ and remove state $\boldsymbol{x}$ together with its incident steps.

Step (1) takes exponential time in the number of atoms. Steps (2.1)-(2.3) can be seen as *reduction rules* that replace an automaton by a smaller one with the same sum over all paths. In the next section we provide similar rules, but at the syntactic level, i.e., rules which act directly on the negotiation diagram, and *not* on the reachability graph. Two of the three rules are straightforward generalizations of (2.1) and (2.3) above, while the third allows us to remove certain useless arcs from a negotiation.

4 Reduction Rules

A *reduction rule*, or just a rule, is a binary relation on the set of negotiations. Given a rule R, we write $\mathcal{N}_1 \xrightarrow{R} \mathcal{N}_2$ for $(\mathcal{N}_1, \mathcal{N}_2) \in R$. A rule R is *correct* if it preserves equivalence, i.e., if $\mathcal{N}_1 \xrightarrow{R} \mathcal{N}_2$ implies $\mathcal{N}_1 \equiv \mathcal{N}_2$. Notice that, in particular, this implies that $\mathcal{N}_1$ is sound if and only if $\mathcal{N}_2$ is sound.

Given a set of rules $\mathcal{R} = \{R_1, \ldots, R_k\}$, we denote by $\mathcal{R}^*$ the reflexive and transitive closure of $R_1 \cup \ldots \cup R_k$. We say that $\mathcal{R}$ is *complete with respect to a class of negotiations* if, for every negotiation $\mathcal{N}$ in the class, there is a negotiation $\mathcal{N}'$ consisting of a single atom such that $\mathcal{N} \xrightarrow{\mathcal{R}^*} \mathcal{N}'$.

We describe rules as pairs of a *guard* and an *action*; $\mathcal{N}_1 \xrightarrow{R} \mathcal{N}_2$ holds if $\mathcal{N}_1$ satisfies the guard and $\mathcal{N}_2$ is a possible result of applying the action to $\mathcal{N}_1$.

Merge Rule. Intuitively, the *merge rule* merges two outcomes with identical next enabled atoms into one single outcome. It corresponds to the rule of step (2.1) in the previous section.

Definition 7. *Merge rule*
Guard: *N contains an atom n with two distinct outcomes $r_1, r_2 \in R_n$, such that $\mathcal{X}(n,a,r_1) = \mathcal{X}(n,a,r_2)$ for every $a \in A_n$.*
Action: *(1) $R_n \leftarrow (R_n \setminus \{r_1, r_2\}) \cup \{r_f\}$, where r_f is a fresh name.*
(2) For all $a \in P_n$: $\mathcal{X}(n,a,r_f) \leftarrow \mathcal{X}(n,a,r_1)$.
(3) $\delta(n,r_f) \leftarrow \delta(n,r_1) \cup \delta(n,r_2)$.

It is easy to see that the merge rule is correct for arbitrary negotiations.

Shortcut Rule. The shortcut rule corresponds to the rule of step (2.3) in the previous section. We need a preliminary definition.

Definition 8. *Given atoms n, n', we say that (n,r)* unconditionally enables *n' if $P_n \supseteq P_{n'}$ and $\mathcal{X}(n,a,r) = \{n'\}$ for every $a \in P_{n'}$.*

Observe that if (n,r) unconditionally enables n' then, for *every* marking $\boldsymbol{x}$ that enables n, the marking $\boldsymbol{x}'$ given by $\boldsymbol{x} \xrightarrow{(n,r)} \boldsymbol{x}'$ enables n'. Moreover, n' can only be disabled by its own occurrence.

The shortcut rule merges the outcomes of two atoms that can occur one after the other into one single outcome with the same effect. Consider the negotiation fragment shown on the left of Figure 2. The guard of the rule will state that n must unconditionally enable n', which is the case. For every outcome of n', say r_1, the action of the rule adds a fresh outcome r_{1f} to n, and modifies the negotiation so that the occurrence of (n, r_{1f}) has the same effect as the occurrence of the sequence $(n,r)(n',r_1)$. In the figure, shortcutting the outcome (n,r) leaves n' without any input arc, and in this case the rule also removes n'. Otherwise we require that at least one input arc of a party $\tilde{a}$ of n' is an arc (i.e., not a proper hyperarc) from some atom $\tilde{n} \neq n$, annotated by $\tilde{r}$. This implies that after the occurrence of $(\tilde{n}, \tilde{r})$, n' is the only atom agent $\tilde{a}$ is ready to engage in.

Definition 9. *Shortcut rule*
Guard: *N contains an atom n with an outcome r, and an atom n', $n' \neq n$, such that (n,r) unconditionally enables n'. Moreover, if $n' \in \mathcal{X}(\tilde{n}, \tilde{a}, \tilde{r})$ for at least one $\tilde{n} \neq n$ with $\tilde{a} \in P_{\tilde{n}}$ and $\tilde{r} \in R_{\tilde{n}}$, then $\{n'\} = \mathcal{X}(\tilde{n}, \tilde{a}, \tilde{r})$ for some $\tilde{n} \neq n$, $\tilde{a} \in P_{\tilde{n}}$, $\tilde{r} \in R_{\tilde{n}}$.*
Action: *(1) $R_n \leftarrow (R_n \setminus \{r\}) \cup \{r'_f \mid r' \in R_{n'}\}$, where r'_f are fresh names.*
(2) For all $a \in P_{n'}$, $r' \in R_{n'}$: $\mathcal{X}(n,a,r'_f) \leftarrow \mathcal{X}(n',a,r')$.
For all $a \in P \setminus P_{n'}$, $r' \in R_{n'}$: $\mathcal{X}(n,a,r'_f) \leftarrow \mathcal{X}(n,a,r)$.
(3) For all $r' \in R_{n'}$: $\langle n, r'_f \rangle \leftarrow \langle n, r \rangle \langle n', r' \rangle$.
(4) If $\mathcal{X}^{-1}(n') = \emptyset$ after (1)-(3), then remove n' from N, where $\mathcal{X}^{-1}(n') = \{(\tilde{n}, \tilde{a}, \tilde{r}) \in T(N) \mid n' \in \mathcal{X}(\tilde{n}, \tilde{a}, \tilde{r})\}$.

Theorem 2. *The shortcut rule is correct.*

Proof. (Sketch) Let $\mathcal{N}_2$ be the result of applying the rule to $\mathcal{N}_1$. The proof is based on the following observation: Each initial occurrence sequence of $\mathcal{N}_2$ can be translated to a corresponding initial occurrence sequence of $\mathcal{N}_1$ by replacing

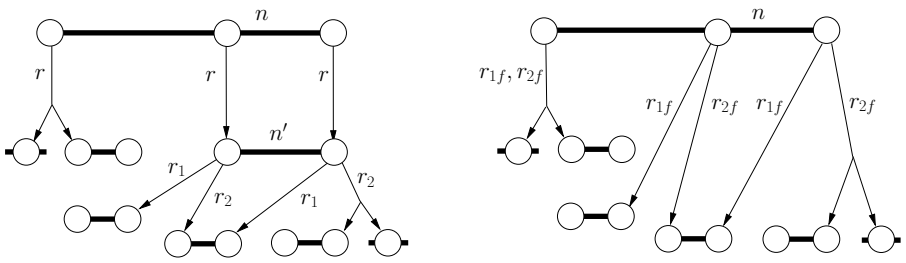


Fig. 2. An application of the shortcut rule

each occurrence of (n, r'_f) by $(n, r)(n', r')$. Conversely, since (n, r) unconditionally enables n', each (n, r)-step of an initial occurrence sequence of $\mathcal{N}_1$ has a corresponding subsequent (n', r')-step, or the occurrence sequence can be extended by a (n', r')-step. This (n', r')-step is independent from all steps in between, i.e., no participant of n' is involved in any of the steps in between. Therefore, the occurrence sequence of $\mathcal{N}_1$ (or its extension by (n', r')) can be reordered to obtain pairs $(n, r)(n', r')$ which can be translated to (n, r'_f)-steps of $\mathcal{N}_2$.

This mutual relation of initial occurrence sequences is used to show that an atom occurs in one negotiation if and only if it occurs in the other one, and that an occurrence sequence of $\mathcal{N}_1$ can be extended by (i.e., is prefix of) another sequence leading to the final marking if the same holds for the corresponding occurrence sequence of $\mathcal{N}_2$, and vice versa.

The full proof requires some delicate case distinctions because n' might still exist in $\mathcal{N}_2$ or not, and in the first case there are occurrences of n' in $\mathcal{N}_2$ that do not belong to pairs $(n, r)(n', r')$. □

Useless Arc Rule. Consider the negotiation on the left of Figure 3, in which all atoms have one outcome r. We have $\mathfrak{X}(n_0, a, r) = \{n_1, n_f\}$, i.e., after the occurrence of (n_0, r) agent a is ready to engage in both n_1 and n_f. However, a always engages in n_1, because the only large step is $(n_0, r)(n_1, r)(n_2, r)(n_f, r)$. In other words, we can set $\mathfrak{X}(n_0, a, n_f) = \{n_1\}$ without changing the behavior. Intuitively, we say that the arc (more precisely, the leg of the hyperarc) leading to the a-port of n_f is useless. The useless arc rule identifies and removes some useless arcs.

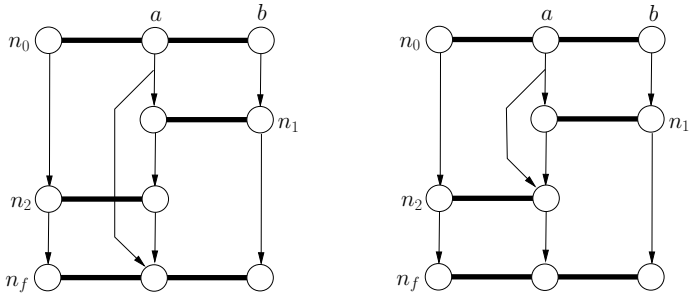


Fig. 3. The useless arc rule can only be applied to the left negotiation

Definition 10. *Useless arc rule.*
Guard: *There are* $(n, a, r), (n, b, r) \in T(N)$ *and two distinct atoms* $n', n'' \in N$ *such that* $a, b \in P_{n'} \cap P_{n''}$, $n', n'' \in \mathcal{X}(n, a, r)$ *and* $\mathcal{X}(n, b, r) = \{n'\}$.
Action: $\mathcal{X}(n, a, r) \leftarrow \mathcal{X}(n, a, r) \setminus \{n''\}$.

The rule can be applied to the negotiation on the left of Figure 3 by instantiating $n := n_0$, $n' := n_1$, and $n'' := n_f$. It cannot be applied to the negotiation on the right. If we set $n := n_0$, $n' := n_1$, and $n'' := n_2$, then $a \notin P_{n_2}$.

Theorem 3. *The useless arc rule is correct.*

Proof. (Sketch) Let $\mathcal{N}_2$ be the result of applying the rule to $\mathcal{N}_1$. We prove that $\mathcal{N}_1$ and $\mathcal{N}_2$ have the same initial occurrence sequences, from which the result easily follows. Let $\mathcal{X}_1, \mathcal{X}_2$ be the transition functions of $\mathcal{N}_1$ and $\mathcal{N}_2$, respectively. Since $\mathcal{X}_2(n, a, r) \subseteq \mathcal{X}_1(n, a, r)$ for every $(n, a, r) \in T(N)$, every initial occurrence sequence of $\mathcal{N}_2$ is also an initial occurrence sequence of $\mathcal{N}_1$. To prove that every initial occurrence sequence σ of $\mathcal{N}_1$ is also an initial occurrence sequence of $\mathcal{N}_2$, assume there is $\sigma = \sigma_1\sigma_2$ such that σ_1 is an initial occurrence sequence of both $\mathcal{N}_1$ and $\mathcal{N}_2$ whereas the first step of σ_2 is only possible in $\mathcal{N}_1$. Since both negotiations only differ w.r.t. $\mathcal{X}(n, a, r)$, this step must be an occurrence of agent n'' occurring after an occurrence of (n, r) such that no other atom of $\mathcal{X}(n, a, r)$ occurred in between. This holds in particular for $n' \in \mathcal{X}(n, a, r)$. But, since $\mathcal{X}(n, b, r) = \{n'\}$ and b participates both in n' and in n'', n' must occur before n'' after (n, r) in the occurrence sequence – a contradiction. □

5 (Weakly) Deterministic Negotiations

Definition 11. *An agent* $a \in A$ *is* deterministic *if for every* $(n, a, r) \in T(N)$ *such that* $n \neq n_f$ *there exists one atom* n' *such that* $\mathcal{X}(n, a, r) = \{n'\}$.

The negotiation $\mathcal{N}$ *is* weakly deterministic *if for every* $(n, a, r) \in T(N)$ *there is a deterministic agent* b *that is a party of every atom in* $\mathcal{X}(n, a, r)$*, i.e.,* $b \in P_{n'}$ *for every* $n' \in \mathcal{X}(n, a, r)$*. It is* deterministic *if all its agents are deterministic.*

Graphically, an agent a is deterministic if no proper hyperarc leaves any port of a. Consider the negotiations of Figure 1. In the acyclic negotiation both Father and Daughter are deterministic, while Mother is not. In the ping-pong negotiation only Daughter is deterministic. Both negotiations are weakly deterministic, because Daughter participates in all atoms, and so can be always chosen as the party b required by the definition. Observe that the notion of deterministic agent does not refer to the behavior of atoms, which is intrinsically nondeterministic with respect to its possible outcomes and even to its state transformations. Rather, it refers to the *composition* of negotiations: For each atom n, the next atom of a deterministic agent is completely determined by the outcome of n.

Weakly deterministic negotiations have a natural semantic justification. Consider a negotiation with two agents a, b and three atoms $\{n_0, n_1, n_f\}$. All atoms have the same parties a, b and one outcome r, such that $\mathcal{X}(n_0, a, r) = \{n_1, n_f\} = \mathcal{X}(n_0, b, r)$ and $\mathcal{X}(n_1, a, r) = \{n_f\} = \mathcal{X}(n_1, b, r)$. After the occurrence of (n_0, r)

the parties a and b are ready to engage in both n_1 and n_f, and so which of them occurs requires a "meta-negotiation" between a and b. This meta-negotiation, however, is not part of the model and, more importantly, it can be difficult to implement, since it requires to break a symmetry. In a weakly deterministic negotiation this situation cannot happen. If $\mathcal{X}(n, a, r)$ contains more than one atom, then some deterministic agent b is a part of all atoms in $\mathcal{X}(n, a, r)$. If some atom of $\mathcal{X}(n, a, r)$ becomes enabled, say n', then because agent b is ready to engage in it, and, since b is deterministic, b is not ready to engage in any other atom. So n' is the only enabled atom of $\mathcal{X}(n, a, r)$, and n' is the negotiation that a will engage in next. This is very easy to implement: b just sends a message to a telling her that she should commit to n'.

For deterministic negotiations, the second part of the guard of the shortcut rule is always satisfied. Using the notation of the shortcut rule, this condition requires that the atom n' is the only atom in $\mathcal{X}(\tilde{n}, \tilde{a}, \tilde{r})$ for some $(\tilde{n}, \tilde{r})$, provided n' is not only in $\mathcal{X}(n, a, r)$ for some $a \in P_n$ (i.e., provided n' is not removed by the application of the rule). This clearly holds if $\tilde{n}$ is deterministic, as all agents are deterministic in deterministic negotiations.

In the next section we show that, on top of their semantic justification, weakly deterministic and deterministic negotiations are also interesting from an analysis point of view. We prove that the shortcut and useless arc rules are complete for acyclic, weakly deterministic negotiations, which of course implies that the same rules plus the merge are complete, too. A second result proves that a polynomial number of applications of the merge and shortcut rules suffices to summarize any sound deterministic acyclic negotiation (the useless arc rule is irrelevant for deterministic negotiations).

6 Completeness and Complexity

We start with the completeness result for the weakly deterministic case.

Theorem 4. *The shortcut and useless arc rules are complete for acyclic, weakly deterministic negotiations.*

Proof. (Sketch) Let $\mathcal{N}$ be a sound and acyclic weakly deterministic negotiation. The proof has two parts.

(1) If $\mathcal{N}$ has more than one atom, then the shortcut rule or the useless arc rule can be applied to it.

Since $\mathcal{N}$ is acyclic, its graph generates a partial order on atoms in the obvious way ($n < n'$ if there is a path from n to n'). Clearly n_0 is the unique minimal element. We choose an arbitrary linearisation of this partial order. Since $\mathcal{N}$ has more than one atom, this linearisation begins with n_0 and has some second element, say n_1.

The full proof shows that, if $\{n_1\} = \mathcal{X}(n_0, a, r_0)$ for every party a of n_1, then the shortcut rule can be applied to n_0, n_1, and if $\{n_1\} \neq \mathcal{X}(n_0, a, r_0)$ for some party a of n_1, then the useless arc rule is applicable.

(2) The shortcut and useless arc rules cannot be applied infinitely often. This follows from two facts: the application of the shortcut rule does not increase the length of any large step of a negotiation, and strictly decreases the length of at least one large step (by replacing a sequence of two outcomes by one single outcome); the application of the useless arc rule does not change the large steps but decreases the number of arcs.

By (2) every maximal sequence of applications of the rule terminates. By (1) it terminates with a negotiation containing one single atom. □

Next we prove that a *polynomial number* of applications of the merge and shortcut rules suffice to summarize any sound *deterministic* acyclic negotiation (SDAN). For this we have to follow a strategy in the application of the shortcut rule.

Definition 12. *The* deterministic shortcut rule*, or* d-shortcut *rule, is the result of adding to the guard of the shortcut rule a new condition: (3) n' has at most one outcome (the actions of the shortcut and d-shortcut rules coincide).*

We say that a SDAN is *irreducible* if neither the merge nor the d-shortcut rule can be applied to it. In the rest of the section we prove that irreducible SDANs are necessarily atomic. The proof, which is rather involved, proceeds in three steps. First, we prove a technical lemma showing that SDANs can be reduced so that all agents participate in every atom with more than one outcome. Then we use this result to prove that, loosely speaking, every SDAN can be reduced to a "replication" of a negotiation with only one agent: if $\mathcal{X}(n, a, r) = \{n'\}$ for some agent a, then $\mathcal{X}(n, a, r) = \{n'\}$ for *every* agent a. In the third step, we show that replications can be reduced to atomic negotiations. Finally, we analyze the number of rule applications needed in each of these three steps, and conclude.

Lemma 1. *Let $\mathcal{N}$ be an irreducible SDAN and let $n \neq n_f$ be an atom of $\mathcal{N}$ with more than one outcome. Then every agent participates in n.*

Proof. (Sketch) We first prove the following key claim: The atom n has an outcome r such that either (n, r) unconditionally enables n_f, or (n, r) unconditionally enables some atom with more than one outcome.
To prove the claim, we first show that it suffices to prove: if some outcome (n, r) unconditionally enables some atom, then the claim holds. For this we assume the contrary, and prove that $\mathcal{N}$ then contains a cycle, contradicting the hypothesis.

By repeated application of the claim we find a chain $(n_1, r_1) \dots (n_k, r_k)$ such that $n_1 = n$, $n_k = n_f$, and (n_i, r_i) unconditionally enables n_{i+1} for every $1 \leq i \leq k - 1$. By the definition of "unconditionally enabled" we have $P_1 \supseteq P_2 \supseteq \cdots \supseteq P_k = P_f$. Since $P_f = A$, we obtain $P_1 = A$. □

Lemma 2. *Let $\mathcal{N}$ be an irreducible SDAN. Every agent participates in every atom, and for every atom $n \neq n_f$ and every outcome r there is an atom n' satisfying $\mathcal{X}(n, a, r) = \{n'\}$ for every agent a.*

Proof. We first show that every agent participates in every atom. By Lemma 1, it suffices to prove that every atom $n \neq n_f$ has more than one outcome. Assume the contrary, i.e., some atom different from n_f has only one outcome. Since, by soundness, every atom can occur, there is an occurrence sequence $(n_0, r_0)(n_1, r_1) \cdots (n_k, r_k)$ such that n_k has only one outcome and all of $n_0, \ldots, n_{k-1}$ have more than one outcome. By Lemma 1, all agents participate in all of n_0, n_1, n_{k-1}. It follows that (n_i, r_i) unconditionally enables (n_{i+1}, r_{i+1}) for every $0 \leq i \leq k-1$. In particular, (n_{k-1}, r_{k-1}) unconditionally enables (n_k, r_k). But then, since n_k only has one outcome, the d-shortcut rule can be applied to n_{k-1}, n, contradicting the hypothesis that $\mathcal{N}$ is irreducible.

For the second part, assume there is an atom $n \neq n_f$, an outcome r of n, and two distinct agents a_1, a_2 such that $\mathcal{X}(n, a_1, r) = \{n_1\} \neq \{n_2\} = \mathcal{X}(n, a_2, r)$. By the first part, every agent participates in n, n_1 and n_2. Since $\mathcal{N}$ is sound, some reachable marking $\boldsymbol{x}$ enables n. Moreover, since all agents participate in n, and $\mathcal{N}$ is deterministic, the marking $\boldsymbol{x}$ only enables n. Let $\boldsymbol{x}'$ be the marking given by $\boldsymbol{x} \xrightarrow{(n,r)} \boldsymbol{x}'$. Since a_1 participates in all atoms, no atom different from n_1 can be enabled at $\boldsymbol{x}'$. Symmetrically, no atom different from n_2 can be enabled at $\boldsymbol{x}'$. So $\boldsymbol{x}'$ does not enable any atom, contradicting that $\mathcal{N}$ is sound. □

Theorem 5. *Let $\mathcal{N}$ be an irreducible SDAN. Then $\mathcal{N}$ contains only one atom.*

Proof. (Sketch) Assume $\mathcal{N}$ contains more than one atom. Fore every atom $n \neq n_f$, let $l(n)$ be the length of the longest path from n to n_f in the graph of $\mathcal{N}$. Let $n_{\min}$ be any atom such that $l(n_{\min})$ is minimal, and let r be an arbitrary outcome of $n_{\min}$. By Lemma 2 there is an atom n' such that $\mathcal{X}(n_{\min}, a, r) = \{n'\}$ for every agent a. If $n' \neq n_f$ then by acyclicity we have $l(n') < l(n_{\min})$, contradicting the minimality of $n_{\min}$. So we have $\mathcal{X}(n_{\min}, a, r) = \{n'\}$ for every outcome r of $n_{\min}$ and every agent a. If $n_{\min}$ has more than one outcome, then the merge rule is applicable. If $n_{\min}$ has one outcome, then, since it unconditionally enables n_f, the d-shortcut rule is applicable. In both cases we get a contradiction to irreducibility. □

Definition 13. *For every atom n and outcome r, let $shoc(n, r)$ be the length of a shortest maximal occurrence sequence containing (n, r) minus 1, and let $Shoc(\mathcal{N}) = \sum_{n \in N, r \in R} shoc(n, r)$. Finally, let $Out(\mathcal{N}) = \sum_{(P,R,\delta) \in N \setminus \{n_f\}} |R|$ be the total number of outcomes of $\mathcal{N}$, excluding those of the final atom.*

Notice that if $\mathcal{N}$ has K atoms then $shoc(n, r) \leq K - 1$ holds for every atom n and outcome r. Further, if $K = 1$ then $Shoc(\mathcal{N}) = 0 = Out(\mathcal{N})$.

Theorem 6. *Every SDAN $\mathcal{N}$ can be completely reduced by means of $Out(\mathcal{N})$ applications of the merge rule and $Shoc(\mathcal{N})$ applications of the d-shortcut rule.*

Proof. Let $\mathcal{N}$ and $\mathcal{N}'$ be negotiations such that $\mathcal{N}'$ is obtained from $\mathcal{N}$ by means of the merge or the d-shortcut rule. For the merge rule we have $Out(\mathcal{N}') < Out(\mathcal{N})$ and $Shoc(\mathcal{N}') \leq Out(\mathcal{N})$ because the rule reduces the number of outcomes by one. For the d-shortcut rule we have $Out(\mathcal{N}') = Out(\mathcal{N})$ because if it is applied to pairs n, n' such that n' has one single outcome, and $Shoc(\mathcal{N}') < Shoc(\mathcal{N})$, because $Shoc(n, r'_f) < Shoc(n, r)$. □

7 Conclusions

We have introduced negotiations, a formal model of concurrency with negotiation atoms as primitive. We have defined and studied two important analysis problems: soundness, which coincides with the soundness notion for workflow nets, and the new *summarization problem*. We have provided a complete set of rules for sound acyclic, weakly deterministic negotiations, and we have shown that the rules allow one to compute a summary of a sound deterministic negotiations in polynomial time.

Several open questions deserve further study. Our results show that summarization can be solved in polynomial time for deterministic, acyclic negotiations, and is co-NP-hard for arbitrary acyclic negotiations. The precise complexity of the weak deterministic case is still open. We are currently working on a generalization of Rule 2.2. of Section 3.2, such that we can completely reduce in polynomial time *arbitrary* deterministic negotiations, even if they contain cycles.

Related Work. Previous work on Petri net analysis by means of reductions has already been discussed in the Introduction.

A number of papers have modelled specific distributed negotiation protocols with the help of Petri nets or process calculi (see [21,19,3]). However, these papers do not address the issue of negotiation as concurrency primitive.

The feature of summarizing parts of a negotiation to single negotiation atoms has several analogies in Petri net theory, among these the concept of zero-safe Petri nets. By abstracting from reachable markings which mark distinguished "zero-places", transactions can be modelled by zero-safe Petri nets [8]. Reference [7] extends this concept to reconfigurable, dynamic high-level Petri nets.

A related line of research studies global types and session types to model multiparty sessions [17]. See [9] for an overview that also covers choreography-based approaches for web services. This research emphasises communication aspects in the formal setting of mobile processes. Thus, the aim differs from our aim. However, it might be worth trying to combine the two approaches.

Finally, the graphical representation of negotiations was partly inspired by the BIP component framework [4,6], where a set of sequential components (i.e., the agents) interact by synchronizing on certain actions (i.e., the atoms).

Acknowledgement. We thank the reviewers, in particular for their hints to related papers. We also thank Stephan Barth, Eike Best, and Jan Kretinsky for very helpful discussions.

References

1. van der Aalst, W.M.P.: The application of Petri nets to workflow management. J. Circuits, Syst. and Comput. 8(1), 21–66 (1998)
2. 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 Asp. Comp. 23(3), 333–363 (2011)

3. Bacarin, E., Madeira, E.R.M., Medeiros, C.B., van der Aalst, W.M.P.: *SpiCa*'s multi-party negotiation protocol: Implementation using YAWL. Int. J. Cooperative Inf. Syst. 20(3), 221–259 (2011)
4. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software 28(3), 41–48 (2011)
5. Berthelot, G.: Transformations and decompositions of nets. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 254, pp. 359–376. Springer, Heidelberg (1987)
6. Bliudze, S., Sifakis, J.: The algebra of connectors - structuring interaction in BIP. IEEE Trans. Computers 57(10), 1315–1330 (2008)
7. Bruni, R., Melgratti, H.C., Montanari, U.: Extending the zero-safe approach to coloured, reconfigurable and dynamic nets. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 291–327. Springer, Heidelberg (2004)
8. Bruni, R., Montanari, U.: Executing transactions in zero-safe nets. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000, pp. 83–102 (2000)
9. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multi-party session. Logical Methods in Computer Science 8(1) (2012)
10. Desel, J.: Reduction and design of well-behaved concurrent systems. In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458, pp. 166–181. Springer, Heidelberg (1990)
11. Desel, J., Esparza, J.: Free choice Petri nets. Cambridge University Press, New York (1995)
12. Desel, J., Esparza, J.: On negotiation as concurrency primitive. Technical report, Technische Universität München, Germany (2013) Available via arxiv.org
13. van Dongen, B.F., van der Aalst, W.M.P., Verbeek, H.M.W.: Verification of EPCs: Using reduction rules and Petri nets. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 372–386. Springer, Heidelberg (2005)
14. Genrich, H.J., Thiagarajan, P.S.: A theory of bipolar synchronization schemes. Theor. Comput. Sci. 30, 241–318 (1984)
15. Haddad, S.: A reduction theory for coloured nets. In: Rozenberg, G. (ed.) APN 1989. LNCS, vol. 424, pp. 209–235. Springer, Heidelberg (1990)
16. Haddad, S., Pradat-Peyre, J.-F.: New efficient Petri nets reductions for parallel programs verification. Parallel Processing Letters 16(1), 101–116 (2006)
17. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) POPL, pp. 273–284. ACM (2008)
18. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2006)
19. Ji, S., Tian, Q., Liang, Y.: A Petri-net-based modeling framework for automated negotiation protocols in electronic commerce. In: Lukose, D., Shi, Z. (eds.) PRIMA 2005. LNCS, vol. 4078, pp. 324–336. Springer, Heidelberg (2009)
20. Papadimitriou, C.H., Yannakakis, M.: The complexity of facets (and some facets of complexity). In: Lewis, H.R., Simons, B.B., Burkhard, W.A., Landweber, L.H. (eds.) STOC, pp. 255–260. ACM (1982)
21. Salaün, G., Ferrara, A., Chirichiello, A.: Negotiation among web services using LOTOS/CADP. In: Zhang, L.-J., Jeckle, M. (eds.) ECOWS 2004. LNCS, vol. 3250, pp. 198–212. Springer, Heidelberg (2004)
22. Verbeek, H.M.W., Wynn, M.T., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Reduction rules for reset/inhibitor nets. J. Comp. Syst. Sci. 76(2), 125–143 (2010)

Satisfiability of CTL* with Constraints*

Claudia Carapelle, Alexander Kartzow, and Markus Lohrey

Institut für Informatik, Universität Leipzig, Germany

Abstract. We show that satisfiability for CTL^* with equality-, order-, and modulo-constraints over $\mathbb{Z}$ is decidable. Previously, decidability was only known for certain fragments of CTL^*, e.g., the existential and positive fragments and EF.

1 Introduction

Temporal logics like LTL, CTL or CTL^* are nowadays standard languages for specifying system properties in model-checking. They are interpreted over node labeled graphs (Kripke structures), where the node labels (also called atomic propositions) represent abstract properties of a system. Clearly, such an abstracted system state does in general not contain all the information of the original system state. Consider for instance a program that manipulates two integer variables x and y. A useful abstraction might be to introduce atomic propositions $v_{-2^{32}}, \ldots, v_{2^{32}}$ for $v \in \{x, y\}$, where the meaning of v_k for $-2^{32} < k < 2^{32}$ is that the variable $v \in \{x, y\}$ currently holds the value k, and $v_{-2^{32}}$ (resp., $v_{2^{32}}$) means that the current value of v is at most -2^{32} (resp., at least 2^{32}). It is evident that such an abstraction might lead to incorrect results in model-checking.

To overcome these problems, extensions of temporal logics with constraints have been studied. Let us explain the idea in the context of LTL. For a fixed relational structure $\mathcal{A}$ (typical examples for $\mathcal{A}$ are number domains like the integers or rationals extended with certain relations) one adds atomic formulas of the form $r(\mathsf{X}^{i_1} x_1, \ldots, \mathsf{X}^{i_k} x_k)$ (so called constraints) to standard LTL. Here, r is (a name of) one of the relations of the structure $\mathcal{A}$, $i_1, \ldots, i_k \geq 0$, and $x_1, \ldots, x_k$ are variables that range over the universe of $\mathcal{A}$. An LTL-formula containing such constraints is interpreted over (infinite) paths of a standard Kripke structure, where in addition every node (state) associates with each of the variables $x_1, \ldots, x_k$ an element of $\mathcal{A}$ (one can think of $\mathcal{A}$-registers attached to the system states). A constraint $r(\mathsf{X}^{i_1} x_1, \ldots, \mathsf{X}^{i_k} x_k)$ holds in a path $s_0 \to s_1 \to s_2 \to \cdots$ if the tuple $(a_1, \ldots, a_k)$, where a_j is the value of variable x_j at state s_{i_j}, belongs to the $\mathcal{A}$-relation r. In this way, the values of variables at different system states can be compared. In our example from the first paragraph, one might choose for $\mathcal{A}$ the structure $(\mathbb{Z}, <, =, (=_a)_{a \in \mathbb{Z}})$, where $=_a$ is the unary predicate that only holds for a. This structure has infinitely many predicates, which is not a problem; our main result will actually talk about an expansion of $(\mathbb{Z}, <, =, (=_a)_{a \in \mathbb{Z}})$. Then, one might for instance write down a formula $(<(x, \mathsf{X}^1 y))\mathsf{U}(=_{100}(y))$ which holds on a path if and only if there is a point of time where variable y holds the value 100 and for all previous points of time t, the value of x at time t is strictly smaller than the value of y at time $t + 1$.

* **Omitted proofs can be found in [4].** This work is supported by the DFG Research Training Group 1763 (QuantLA). The second author is supported by the DFG research project GELO.

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 455–469, 2013.

In [9], Demri and Gascon studied LTL extended with constraints from a language IPC*. If we disregard succinctness aspects, these constraints are equivalent to constraints over the structure

$$\mathcal{Z} = (\mathbb{Z}, <, =, (=_a)_{a\in\mathbb{Z}}, (\equiv_{a,b})_{0\le a<b}), \tag{1}$$

where $=_a$ denotes the unary relation $\{a\}$ and $\equiv_{a,b}$ denotes the unary relation $\{a + xb \mid x \in \mathbb{Z}\}$ (expressing that an integer is congruent to a modulo b). The main result from [9] states that satisfiability of LTL with constraints from $\mathcal{Z}$ is decidable and in fact PSPACE-complete, and hence has the same complexity as satisfiability for LTL without constraints. We should remark that the PSPACE upper bound from [9] even holds for the succinct IPC*-representation of constraints used in [9].

In the same way as outlined for LTL above, constraints can be also added to CTL and CTL* (then, constraints $r(\mathsf{X}^{i_1}x_1, \ldots, \mathsf{X}^{i_k}x_k)$ are path formulas). A weak form of CTL* with constraints from $\mathcal{Z}$ (where only integer variables and the same state can be compared) was first introduced in [5], where it is used to describe properties of infinite transition systems, represented by relational automata. There it is shown that the model checking problem for CTL* over relational automata is undecidable.

Demri and Gascon [9] asked whether satisfiability of CTL* with constraints from $\mathcal{Z}$ over Kripke structures is decidable. This problem was investigated in [3,10], where several partial results where shown: If we replace in $\mathcal{Z}$ the binary predicate $<$ by unary predicates $<_c = \{x \mid x < c\}$ for $c \in \mathbb{Z}$, then satisfiability for CTL* is decidable by [10]. While, for the full structure $\mathcal{Z}$ satisfiability is decidable for the CTL* fragment CEF$^+$ (which contains the existential and universal fragment of CTL* as well as EF) [3].

In this paper we prove that CTL* with constraints over $\mathcal{Z}$ is decidable. Our proof is divided into two steps. The first step provides a tool to prove decidability of CTL* with constraints over any structure $\mathcal{A}$ over a countable (finite or infinite) signature $\mathcal{S}$ (the structure $\mathcal{A}$ has to satisfy the additional property that the complement of any of its relations has to be definable in positive existential first-order logic over $\mathcal{A}$). Let $\mathcal{L}$ be a logic that satisfies the following two properties: (i) satisfiability of a given $\mathcal{L}$-sentence over the class of infinite node-labeled trees is decidable, and (ii) $\mathcal{L}$ is closed under boolean combinations with monadic second-order formulas (MSO). A typical such logic is MSO itself. By Rabin's seminal tree theorem, satisfiability of MSO-sentences over infinite node-labeled trees is decidable. Assuming $\mathcal{L}$ has these two properties, we prove that satisfiability of CTL* with constraints over $\mathcal{A}$ is decidable if one can compute from a given finite subsignature $\sigma \subseteq \mathcal{S}$ an $\mathcal{L}$-sentence ψ_σ (over the signature σ) such that for every countable σ-structure $\mathcal{B}$: $\mathcal{B} \models \psi_\sigma$ if and only if there exists a homomorphism from $\mathcal{B}$ to $\mathcal{A}$ (i.e., a mapping from the domain of $\mathcal{B}$ to the domain of $\mathcal{A}$ that preserves all relations from σ). We say that the structure $\mathcal{A}$ has the property EHomDef($\mathcal{L}$) if such a computable function $\sigma \mapsto \psi_\sigma$ exists. EHomDefstands for "existence of homomorphism is definable". For instance, the structure $(\mathbb{Q}, <, =)$ has the property EHomDef(MSO), see Example 7.

It is not clear whether $\mathcal{Z}$ from (1) has the property EHomDef(MSO) (we conjecture that it does not). Hence, we need a different logic. It turns out that $\mathcal{Z}$ has the property EHomDef(WMSO+B), where WMSO+B is the extension of weak monadic second-order logic (where only quantification over finite subsets is allowed) with the bounding

quantifier B. A formula $\mathsf{B}X : \varphi$ holds in a structure $\mathcal{A}$ if and only if there exists a bound $b \in \mathbb{N}$ such that for every finite subset B of the domain of $\mathcal{A}$ with $\mathcal{A} \models \varphi(B)$ we have $|B| \leq b$. Recently, Bojańczyk and Toruńczyk have shown that satisfiability of WMSO+B over infinite node-labeled trees is decidable [1]. The next problem is that WMSO+B is not closed under boolean combinations with MSO-sentences. But fortunately, the decidability proof for WMSO+B can be extended to boolean combinations of MSO-sentences and (WMSO+B)-sentences, see Section 3 for details. This finally shows that satisfiability of CTL* with constraints from $\mathcal{Z}$ is decidable.

While it would be extremely useful to add successor constraints ($y = x + 1$) to $\mathcal{Z}$, this would lead to undecidability even for LTL [8] and the very basic description logic $\mathcal{ALC}$ [13], which is basically multi-modal logic. Nonetheless $\mathcal{Z}$ allows qualitative representation of increment, for example $x = y + 1$ can be abstracted by $(y > x) \wedge (\equiv_{1,2^k} (y))$ where k is a large natural number. This is why temporal logics extended with constraints over $\mathcal{Z}$ seem to be a good compromise between (unexpressive) total abstraction and (undecidable) high concretion.

In the area of knowledge representation, extensions of description logics with constraints from so called concrete domains have been intensively studied, see [11] for a survey. In [12], it was shown that the extension of the description logic $\mathcal{ALC}$ with constraints from $(\mathbb{Q}, <, =)$ has a decidable (EXPTIME-complete) satisfiability problem with respect to general TBoxes (also known as general concept inclusions). Such a TBox can be seen as a second $\mathcal{ALC}$-formula that has to hold in all nodes of a model. Our decidability proof is partly inspired by the construction from [12], which in contrast to our proof is purely automata-theoretic. Further results for description logics and concrete domains can be found in [13,14].

Unfortunately, our proof does not yield any complexity bound for satisfiability of CTL* with constraints from $\mathcal{Z}$. The boolean combinations of (WMSO+B)-sentences and MSO sentences that have to be checked for satisfiability (over infinite trees) are of a simple structure, in particular their quantifier depth is not high. But no complexity statement for satisfiability of WMSO+B is made in [1], and it seems to be difficult to analyze the algorithm from [1] (but it seems to be elementary for a fixed quantifier depth). It is based on a construction for cost functions over finite trees from [6], where the authors only note that their construction seems to have very high complexity.

2 Preliminaries

Let $[1, d] = \{1, \ldots, d\}$. For a word $w = a_1 a_2 \cdots a_l \in [1, d]^*$ and $k \leq l$ we define $w[: k] = a_1 a_2 \cdots a_k$; it is the prefix of w of length k.

Let P be a countable set of (atomic) propositions. A Kripke structure over P is a triple $\mathcal{K} = (D, \to, \rho)$, where (i) D is an arbitrary set of nodes (or states), (ii) $\to$ is a binary relation on D such that for every $u \in D$ there exists $v \in D$ with $u \to v$, and (iii) $\rho : D \to 2^{\mathsf{P}}$ assigns to every node the set of propositions that hold in the node. We require that $\bigcup_{v \in D} \rho(v)$ is finite, i.e., only finitely many propositions appear in $\mathcal{K}$. A $\mathcal{K}$-*path* is an infinite sequence $\pi = (v_0, v_1, v_2, \ldots)$ such that $v_i \to v_{i+1}$ for all $i \geq 0$. For $i \geq 0$ we define the state $\pi(i) = v_i$ and the path $\pi^i = (v_i, v_{i+1}, v_{i+2}, \ldots)$. A *Kripke d-tree* is a Kripke structure of the form $\mathcal{K} = ([1, d]^*, \to, \rho)$, where $\to$ contains all pairs

(u, ui) with $u \in [1,d]^*$ and $1 \leq i \leq d$, i.e., $([1,d]^*, \to)$ is a tree with root ε where every node has d children.

A signature is a countable (finite or infinite) set $\mathcal{S}$ of relation symbols. Every relation symbol $r \in \mathcal{S}$ has an associated arity $\mathsf{ar}(r) \geq 1$. An $\mathcal{S}$-structure is a pair $\mathcal{A} = (A, I)$, where A is a non-empty set and I maps every $r \in \mathcal{S}$ to an $\mathsf{ar}(r)$-ary relation over A. Quite often, we will identify the relation $I(r)$ with the relation symbol r, and we will specify an $\mathcal{S}$-structure as $(A, r_1, r_2, \ldots)$ where $\mathcal{S} = \{r_1, r_2, \ldots\}$. The $\mathcal{S}$-structure $\mathcal{A} = (A, I)$ is *negation-closed* if there exists a computable function that maps a relation symbol $r \in \mathcal{S}$ to a positive existential first-order formula $\varphi_r(x_1, \ldots, x_{\mathsf{ar}(r)})$ (i.e., a formula that is built up from atomic formulas using $\wedge$, $\vee$, and $\exists$) such that $A^{\mathsf{ar}(r)} \setminus I(r) = \{(a_1, \ldots, a_{\mathsf{ar}(r)}) \mid \mathcal{A} \models \varphi_r(a_1, \ldots, a_{\mathsf{ar}(r)})\}$. In other words, the complement of every relation $I(r)$ must be effectively definable by a positive existential first-order formula.

Example 1. The structure $\mathcal{Z}$ from (1) is negation-closed (we will write $x = a$ instead of $=_a(x)$ and similarly for $\equiv_{a,b}$). We have for instance:

- $x \neq y$ if and only if $x < y$ or $y < x$.
- $x \neq a$ if and only if $\exists y \in \mathbb{Z} : y = a \wedge (x < y \vee y < x)$.
- $x \not\equiv a \bmod b$ if and only if $x \equiv c \bmod b$ for some $0 \leq c < b$ with $a \neq c$.

For a subsignature $\sigma \subseteq \mathcal{S}$, a σ-structure $\mathcal{B} = (B, J)$ and an $\mathcal{S}$-structure $\mathcal{A} = (A, I)$, a *homomorphism* $h : \mathcal{B} \to \mathcal{A}$ is a mapping $h : B \to A$ such that for all $r \in \sigma$ and all tuples $(b_1, \ldots, b_{\mathsf{ar}(r)}) \in J(r)$ we have $(h(b_1), \ldots, h(b_{\mathsf{ar}(r)})) \in I(r)$. We write $\mathcal{B} \preceq \mathcal{A}$ if there is a homomorphism from $\mathcal{B}$ to $\mathcal{A}$.

3 MSO and WMSO+B

Recall that *monadic second-order logic* (MSO) is the extension of first-order logic where also quantification over subsets of the underlying structure is allowed. We assume that the reader has some familiarity with MSO. *Weak monadic second-order logic* (WMSO) has the same syntax as MSO but second-order variables only range over finite subsets of the underlying structure. Finally, WMSO+B is the extension of WMSO by the additional quantifier $\mathsf{B}X : \varphi$ (the *bounding quantifier*). The semantics of $\mathsf{B}X : \varphi$ in the structure $\mathcal{A} = (A, I)$ is defined as follows: $\mathcal{A} \models \mathsf{B}X : \varphi(X)$ if and only if there is a bound $b \in \mathbb{N}$ such that $|B| \leq b$ for every finite subset $B \subseteq A$ with $\mathcal{A} \models \varphi(B)$.

Example 2. For later use, we state some example formulas. Let $\varphi(x, y)$ be a WMSO-formula with two free first-order variables x and y. Let $\mathcal{A} = (A, I)$ be a structure and let $E = \{(a, b) \in A \times A \mid \mathcal{A} \models \varphi(a, b)\}$ be the binary relation defined by $\varphi(x, y)$. We define the WMSO-formula $\mathsf{reach}_\varphi(a, b)$ to be

$$\exists X \, \forall Y \big(a \in Y \wedge \forall x \forall y ((x \in Y \wedge y \in X \wedge \varphi(x, y)) \to y \in Y) \to b \in Y \big)$$

It is straightforward to prove that $\mathcal{A} \models \mathsf{reach}_\varphi(a, b)$ if and only if $(a, b) \in E^*$. Note that reach_φ is the standard MSO-formula for reachability but restricted to some finite induced subgraph. Clearly, b is reachable from a in the graph (A, E) if and only if it is in some finite subgraph of (A, E).

Let $\mathsf{ECycle}_\varphi = \exists x\, \exists y(\mathsf{reach}_\varphi(x,y) \wedge \varphi(y,x))$ be the WMSO-formula expressing that there is a cycle in (A,E).

Given a second-order variable Z, we define $\mathsf{reach}^Z_\varphi(a,b)$ to be

$$a \in Z \wedge \forall Y \subseteq Z\big(a \in Y \wedge \forall x \forall y((x \in Y \wedge y \in Z \wedge \varphi(x,y)) \rightarrow y \in Y) \rightarrow b \in Y\big).$$

We have $\mathcal{A} \models \mathsf{reach}^Z_\varphi(a,b)$ iff b is reachable from a in the subgraph of (A,E) induced by the (finite) set Z. Note that $\mathcal{A} \models \mathsf{reach}^Z_\varphi(a,b)$ implies $\{a,b\} \subseteq Z$.

For the next examples we restrict our attention the case that the graph (A,E) defined by $\varphi(x,y)$ is acyclic. Hence, the reflexive transitive closure E^* is a partial order on A. Note that a finite set $F \subseteq A$ is an E-path from $a \in F$ to $b \in F$ if and only if $(F, (E \cap (F \times F))^*)$ is a finite linear order with all elements between a and b. Define the WMSO-formula $\mathsf{Path}_\varphi(a,b,Z)$ as

$$\forall x \in Z\ \forall y \in Z\ (\mathsf{reach}^Z_\varphi(x,y) \vee \mathsf{reach}^Z_\varphi(y,x))\ \wedge \mathsf{reach}^Z_\varphi(a,x) \wedge \mathsf{reach}^Z_\varphi(x,b).$$

For every acyclic (A,E) we have $\mathcal{A} \models \mathsf{Path}_\varphi(a,b,P)$ if and only if P contains exactly the nodes along an E-path from a to b.

We finally define the WMSO+B-formula $\mathsf{BPaths}_\varphi(x,y) = \mathsf{B}Z : \mathsf{Path}_\varphi(x,y,Z)$. By definition of the quantifier B, if (A,E) is acyclic, then $\mathcal{A} \models \mathsf{BPaths}_\varphi(a,b)$ if and only if there is a bound $k \in \mathbb{N}$ on the length of any E-path from a to b.

Next, let $\mathsf{Bool}(\mathsf{MSO}, \mathsf{WMSO}{+}\mathsf{B})$ be the set of all Boolean combinations of MSO-formulas and (WMSO+B)-formulas. We will use the following result.

Theorem 3 (cf. [1]). *One can decide whether for a given $d \in \mathbb{N}$ and a formula $\varphi \in \mathsf{Bool}(\mathsf{MSO}, \mathsf{WMSO}{+}\mathsf{B})$ there exists a Kripke d-tree $\mathcal{K}$ such that $\mathcal{K} \models \varphi$.*

Proof. This theorem follows from results of Bojańczyk and Toruńczyk [1,2]. They introduced puzzles which can be seen as pairs $P = (A,C)$, where A is a parity tree automaton and C is an unboundedness condition C which specifies a certain set of infinite paths labeled by states of A. A puzzle accepts a tree $\mathcal{T}$ if there is an accepting run ρ of A on $\mathcal{T}$ such that for each infinite path π occurring in ρ, $\pi \in C$ holds. In particular, ordinary parity tree automata can be seen as puzzles with trivial unboundedness condition. The proof of our theorem combines the following results.

Lemma 4 ([1]). *From a given (WMSO+B)-formula φ and $d \in \mathbb{N}$ one can construct a puzzle P_φ such that φ is satisfied by some Kripke d-tree iff P_φ is nonempty.*

Lemma 5 ([1]). *Emptiness of puzzles is decidable.*

Lemma 6 (Lemma 17 of [2]). *Puzzles are effectively closed under intersection.*

Let $\varphi \in \mathsf{Bool}(\mathsf{MSO}, \mathsf{WMSO}{+}\mathsf{B})$. First, φ can be effectively transformed into a disjunction $\bigvee_{i=1}^n (\varphi_i \wedge \psi_i)$ where $\varphi_i \in \mathsf{MSO}$ and $\psi_i \in \mathsf{WMSO}{+}\mathsf{B}$ for all i. By Lemma 4, we can construct a puzzle P_i for ψ_i. It is known that the MSO-formula φ_i can be translated into a parity tree automaton A_i. Let P'_i be a puzzle recognizing the intersection of P_i and A_i (cf. Lemma 6). Now φ is satisfiable over Kripke d-trees if and only if there is an i such that $\varphi_i \wedge \psi_i$ is satisfiable over Kripke d-trees if and only if there is an i such that P'_i is nonempty. By Lemma 5, the latter condition is decidable which concludes the proof of the theorem. □

Let $\mathcal{L}$ be a logic (e.g. MSO or Bool(MSO, WMSO+B)). An $\mathcal{S}$-structure $\mathcal{A}$ has the property EHomDef($\mathcal{L}$) (existence of homomorphisms to $\mathcal{A}$ is $\mathcal{L}$-definable) if there is a computable function that maps a finite subsignature $\sigma \subseteq \mathcal{S}$ to an $\mathcal{L}$-sentence φ_σ such that for every countable σ-structure $\mathcal{B}$: $\mathcal{B} \preceq \mathcal{A}$ if and only if $\mathcal{B} \models \varphi_\sigma$.

Example 7. The structure $\mathcal{Q} = (\mathbb{Q}, <, =)$ has the property EHomDef(WMSO) (and EHomDef(MSO)). In [12] it is implicitly shown that for a countable $\{<, =\}$-structure $\mathcal{B} = (B, I)$, $\mathcal{B} \preceq \mathcal{Q}$ if and only if there does not exist $(a, b) \in I(<)$ such that $(b, a) \in (I(<) \cup I(=) \cup I(=)^{-1})^*$. This condition can be easily expressed in WMSO using the reach-construction from Example 2. Note that $I(=)$ is not required to be the identity relation on B.

4 CTL* with Constraints

Let us fix a countably infinite set of atomic propositions P and a countably infinite set of variables V for the rest of the paper. Let $\mathcal{S}$ be a signature. We define an extension of CTL* with constraints over the signature $\mathcal{S}$. We define CTL*($\mathcal{S}$)-state formulas φ and CTL*($\mathcal{S}$)-path formulas ψ by the following grammar, where $p \in \mathsf{P}$, $r \in \mathcal{S}$, $k = \mathsf{ar}(r)$, $i_1, \ldots, i_k \geq 0$, and $x_1, \ldots, x_k \in \mathsf{V}$:

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid \mathsf{E}\psi$$
$$\psi ::= \varphi \mid \neg\psi \mid (\psi \wedge \psi) \mid \mathsf{X}\psi \mid \psi \mathsf{U} \psi \mid r(\mathsf{X}^{i_1} x_1, \ldots, \mathsf{X}^{i_k} x_k)$$

A formula of the form $R := r(\mathsf{X}^{i_1} x_1, \ldots, \mathsf{X}^{i_k} x_k)$ is also called an *atomic constraint* and we define $d(R) = \max\{i_1, \ldots, i_k\}$ (the depth of R). The syntactic difference between CTL*($\mathcal{S}$) and ordinary CTL* lies in the presence of atomic constraints.

Formulas of CTL*($\mathcal{S}$) are interpreted over triples $\mathcal{C} = (\mathcal{A}, \mathcal{K}, \gamma)$, where $\mathcal{A} = (A, I)$ is an $\mathcal{S}$-structure (also called the *concrete domain*), $\mathcal{K} = (D, \to, \rho)$ is a Kripke structure over P, and $\gamma : D \times \mathsf{V} \to A$ assigns to every $(v, x) \in D \times \mathsf{V}$ a value $\gamma(v, x)$ (the value of variable x at node v). We call such a triple $\mathcal{C} = (\mathcal{A}, \mathcal{K}, \gamma)$ an *$\mathcal{A}$-constraint graph*. An $\mathcal{A}$-constraint graph $\mathcal{C} = (\mathcal{A}, \mathcal{K}, \gamma)$ is an *$\mathcal{A}$-constraint d-tree* if $\mathcal{K}$ is a Kripke d-tree.

We now define the semantics of CTL*($\mathcal{S}$). For an $\mathcal{A}$-constraint graph $\mathcal{C} = (\mathcal{A}, \mathcal{K}, \gamma)$ with $\mathcal{A} = (A, I)$ and $\mathcal{K} = (D, \to, \rho)$, a state $v \in D$, a $\mathcal{K}$-path π, a state formula φ, and a path formula ψ we write $(\mathcal{C}, v) \models \varphi$ if φ holds in $(\mathcal{C}, v)$ and $(\mathcal{C}, \pi) \models \psi$ if ψ holds in $(\mathcal{C}, \pi)$. This is inductively defined as follows (for the boolean connectives $\neg$ and $\wedge$ the definitions are as usual and we omit them):

- $(\mathcal{C}, v) \models p$ iff $p \in \rho(v)$.
- $(\mathcal{C}, v) \models \mathsf{E}\psi$ iff there is a $\mathcal{K}$-path π with $\pi(0) = v$ and $(\mathcal{C}, \pi) \models \psi$.
- $(\mathcal{C}, \pi) \models \varphi$ iff $(\mathcal{C}, \pi(0)) \models \varphi$.
- $(\mathcal{C}, \pi) \models \mathsf{X}\psi$ iff $(\mathcal{C}, \pi^1) \models \psi$.
- $(\mathcal{C}, \pi) \models \psi_1 \mathsf{U} \psi_2$ iff there exists $i \geq 0$ such that $(\mathcal{C}, \pi^i) \models \psi_2$ and for all $0 \leq j < i$ we have $(\mathcal{C}, \pi^j) \models \psi_1$.
- $(\mathcal{C}, \pi) \models r(\mathsf{X}^{i_1} x_1, \ldots, \mathsf{X}^{i_n} x_n)$ iff $(\gamma(\pi(i_1), x_1), \ldots, \gamma(\pi(i_n), x_n)) \in I(r)$.

Note that the role of the concrete domain $\mathcal{A}$ and of the valuation function γ is restricted to the semantic of atomic constraints. CTL^*-formulas are interpreted over Kripke structures, and to obtain their semantics it is sufficient to replace $\mathcal{C}$ by $\mathcal{K}$ in the rules above and to remove the last line.

We use the usual abbreviations: $\theta_1 \vee \theta_2 := \neg(\neg\theta_1 \wedge \neg\theta_2)$ (for both state and path formulas), $\mathsf{A}\psi := \neg\mathsf{E}\neg\psi$ (universal path quantifier), $\psi_1 \mathsf{R} \psi_2 := \neg(\neg\psi_1 \mathsf{U} \neg\psi_2)$ (the release operator). Note that $(\mathcal{C}, \pi) \models \psi_1 \mathsf{R} \psi_2$ iff $((\mathcal{C}, \pi^i) \models \psi_2$ for all $i \geq 0$ or there exists $i \geq 0$ such that $(\mathcal{C}, \pi^i) \models \psi_1$ and $(\mathcal{C}, \pi^j) \models \psi_2$ for all $0 \leq j \leq i)$.

Using this extended set of operators we can put every formula into a semantically equivalent *negation normal form*, where $\neg$ only occurs in front of atomic propositions or atomic constraints. Let $\#_{\mathsf{E}}(\theta)$ be the the number of different subformulas of the form $\mathsf{E}\psi$ in the $\mathsf{CTL}^*(\mathcal{S})$-formula θ. Then $\mathsf{CTL}^*(\mathcal{S})$ has the following tree model property:

Theorem 8 (cf. [10]). *Let φ be a $\mathsf{CTL}^*(\mathcal{S})$-state formula in negation normal form and let $\mathcal{A} = (A, I)$ be an $\mathcal{S}$-structure. Then φ is $\mathcal{A}$-satisfiable if and only if there exists an $\mathcal{A}$-constraint $(\#_{\mathsf{E}}(\varphi) + 1)$-tree $\mathcal{C}$ with $(\mathcal{C}, \varepsilon) \models \varphi$.*

Note that for checking $(\mathcal{A}, \mathcal{K}, \gamma) \models \varphi$ we may ignore all propositions $p \in \mathsf{P}$ that do not occur in φ. Similarly, only those values $\gamma(u, x)$, where x is a variable that appears in φ, are relevant. Hence, if V_φ is the finite set of variables that occur in φ, then we can consider γ as a mapping from $D \times \mathsf{V}_\varphi$ to the domain of $\mathcal{A}$. Intuitively, we assign to each node $u \in D$ registers that store the values $\gamma(u, x)$ for $x \in \mathsf{V}_\varphi$.

5 Satisfiability of Constraint CTL^* over a Concrete Domain

When we talk about satisfiability for $\mathsf{CTL}^*(\mathcal{S})$ our setting is as follows: We fix a concrete domain $\mathcal{A} = (A, I)$. Given a $\mathsf{CTL}^*(\mathcal{S})$-state formula φ, we say that φ is $\mathcal{A}$-satisfiable if there is an $\mathcal{A}$-constraint graph $\mathcal{C} = (\mathcal{A}, \mathcal{K}, \gamma)$ and a node v of $\mathcal{K}$ such that $(\mathcal{C}, v) \models \varphi$. With $\mathsf{SATCTL}^*(\mathcal{A})$ we denote the following computational problem: *Is a given state formula $\varphi \in \mathsf{CTL}^*(\mathcal{S})$ $\mathcal{A}$-satisfiable?* The main result of this section is:

Theorem 9. *Let $\mathcal{A}$ be a negation-closed $\mathcal{S}$-structure, which moreover has the property* $\mathsf{EHomDef}(\mathsf{Bool}(\mathsf{MSO}, \mathsf{WMSO}{+}\mathsf{B}))$. *Then the problem* $\mathsf{SATCTL}^*(\mathcal{A})$ *is decidable.*

We say that a $\mathsf{CTL}^*(\mathcal{S})$-formula φ is in *strong negation normal form* if negations only occur in front of atomic propositions (i.e., φ is in negation normal form and there is no subformula $\neg R$ where R is an atomic constraint).

Let us fix a $\mathsf{CTL}^*(\mathcal{S})$-state formula φ in negation normal form and a negation-closed $\mathcal{S}$-structure $\mathcal{A}$ for the rest of this section. We want to check whether φ is $\mathcal{A}$-satisfiable. First, we reduce to formulas in strong negation normal form:

Lemma 10. *Let $\mathcal{A} = (A, I)$ be a negation-closed $\mathcal{S}$-structure. From a given $\mathsf{CTL}^*(\mathcal{S})$-state formula φ one can compute a $\mathsf{CTL}^*(\mathcal{S})$-state formula $\hat{\varphi}$ in strong negation normal form such that φ is $\mathcal{A}$-satisfiable iff $\hat{\varphi}$ is $\mathcal{A}$-satisfiable.*

From now on let us assume that φ is in strong negation normal form. Let $d = \#_{\mathsf{E}}(\varphi)+1$. Let $R_1, \ldots, R_n$ be a list of all atomic constraints that are subformulas of φ, and let V_φ

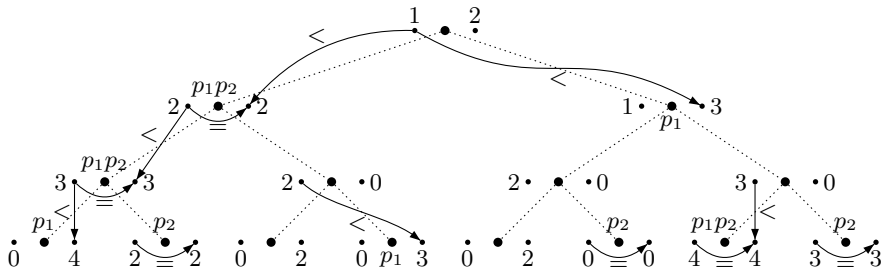


Fig. 1. The $(\mathbb{N}, <, =)$-constraint 2-tree $\mathcal{C}$ from Ex. 11, the Kripke 2-tree $\mathcal{T} = \mathcal{C}^a$, and the structure $\mathcal{G}_\mathcal{T}$

be the finite set of variables that occur in φ. Let us fix new propositions $p_1, \ldots, p_n$ (one for each R_i) that do not occur in φ. Let $d_i = d(R_i)$ be the depth of the constraint R_i. We denote with φ^a the (ordinary) CTL^*-formula obtained from φ by replacing every occurrence of a constraint R_i by $\mathsf{X}^{d_i} p_i$. Given an $\mathcal{A}$-constraint d-tree $\mathcal{C} = (\mathcal{A}, \mathcal{K}, \gamma)$, where $\mathcal{K} = ([1,d]^*, \to, \rho)$ and $\rho(v) \cap \{p_1, \ldots, p_n\} = \emptyset$ for all $v \in [1,d]^*$, we define a Kripke d-tree $\mathcal{C}^a = ([1,d]^*, \to, \rho^a)$, where $\rho^a(v)$ contains

- all propositions from $\rho(v)$ and
- all propositions p_i $(1 \leq i \leq n)$ such that the following holds, where we assume that R_i has the form $r(\mathsf{X}^{j_1} x_1, \ldots, \mathsf{X}^{j_k} x_k)$ with $k = \mathsf{ar}(r)$ (hence, $d_i = \max\{j_1, \ldots, j_k\}$):
  - $v = su$ with $|u| = d_i$
  - $(\gamma(su_1, x_1), \ldots, \gamma(su_k, x_k)) \in I(r)$, where $u_l = u[: j_l]$ for $1 \leq l \leq k$.

Hence, the fact that proposition p_i labels node su with $|u| = d_i$ means that the constraint R_i holds along every path that starts in node s and descends in the tree down via node su. The superscript "a" in $\mathcal{C}^a$ stands for "abstracted" since we abstract from the concrete constraints and replace them by new propositions.

Moreover, given a Kripke d-tree $\mathcal{T} = ([1,d]^*, \to, \rho)$ (where the new propositions $p_1, \ldots, p_n$ are allowed to occur in $\mathcal{T}$) we define a countable $\mathcal{S}$-structure $\mathcal{G}_\mathcal{T} = ([1,d]^* \times \mathsf{V}_\varphi, J)$ as follows: The interpretation $J(r)$ of the relation symbol $r \in \mathcal{S}$ contains all k-tuples (where $k = \mathsf{ar}(r)$) $((su_1, x_1), \ldots, (su_k, x_k))$ for which there exist $1 \leq i \leq n$ and $u \in [1,d]^*$ with $|u| = d_i$ such that $p_i \in \rho(su)$, $R_i = r(\mathsf{X}^{j_1} x_1, \ldots, \mathsf{X}^{j_k} x_k)$, and $u_t = u[: j_t]$ for $1 \leq t \leq k$.

Example 11. Figure 1 shows an example, where we assume that $d = 2$ and $n = 2$, $R_1 = [<(x_1, \mathsf{X}x_2)]$, and $R_2 = [=(\mathsf{X}x_1, \mathsf{X}x_2)]$. The figure shows an initial part of an $(\mathbb{N}, <, =)$-constraint 2-tree $\mathcal{C} = ((\mathbb{N}, <, =), \mathcal{K}, \gamma)$. The edges of the Kripke 2-tree $\mathcal{K}$ are dotted. We assume that $\mathcal{K}$ is defined over the empty set of propositions. The node to the left (resp., right) of a tree node u is labeled by the value $\gamma(u, x_1)$ (resp. $\gamma(u, x_2)$). The figure shows the labeling of tree nodes with the two new propositions p_1 and p_2 (corresponding to R_1 and R_2) as well as the $\{<, =\}$-structure $\mathcal{G}_\mathcal{T}$ for $\mathcal{T} = \mathcal{C}^a$.

Lemma 12. *Let φ be a $\mathsf{CTL}^*(\mathcal{S})$-state formula in strong negation normal form. The formula φ is $\mathcal{A}$-satisfiable if and only if there exists a Kripke $(\#_{\mathsf{E}}(\varphi)+1)$-tree $\mathcal{T}$ such that $(\mathcal{T},\varepsilon) \models \varphi^a$ and $\mathcal{G}_\mathcal{T} \preceq \mathcal{A}$.*

Let $\theta = \varphi^a$ for the further discussion. Hence, θ is an ordinary CTL^*-state formula, where negations only occur in front of propositions from $\mathsf{P} \setminus \{p_1,\ldots,p_m\}$, and $d = \#_{\mathsf{E}}(\theta)+1$. By Lemma 12, we have to check, whether there exists a Kripke d-tree $\mathcal{T}$ such that $(\mathcal{T},\varepsilon) \models \theta$ and $\mathcal{G}_\mathcal{T} \preceq \mathcal{A}$.

Let $\sigma \subseteq \mathcal{S}$ be the finite subsignature consisting of all predicate symbols that occur in our initial $\mathsf{CTL}^*(\mathcal{S})$-formula φ. Note that $\mathcal{G}_\mathcal{T}$ is actually a σ-structure. Since the concrete domain $\mathcal{A}$ has the property $\mathsf{EHomDef}(\mathsf{Bool}(\mathsf{MSO},\mathsf{WMSO}{+}\mathsf{B}))$, one can compute from σ a $\mathsf{Bool}(\mathsf{MSO},\mathsf{WMSO}{+}\mathsf{B})$-formula α such that for every countable σ-structure $\mathcal{B}$ we have $\mathcal{B} \models \alpha$ if and only if $\mathcal{B} \preceq \mathcal{A}$. Hence, our new goal is to decide, whether there exists a Kripke d-tree $\mathcal{T}$ such that $(\mathcal{T},\varepsilon) \models \theta$ and $\mathcal{G}_\mathcal{T} \models \alpha$ (note that $\mathcal{G}_\mathcal{T}$ is countable). It is well known that every CTL^*-state formula can be effectively transformed into an equivalent MSO-formula with a single free first-order variable. Since the root ε of a tree is first-order definable, we get an MSO-sentence ψ such that $(\mathcal{T},\varepsilon) \models \theta$ if and only if $\mathcal{T} \models \psi$. Hence, we have to check whether there exists a Kripke d-tree $\mathcal{T}$ such that $\mathcal{T} \models \psi$ and $\mathcal{G}_\mathcal{T} \models \alpha$. If we can translate the $\mathsf{Bool}(\mathsf{MSO},\mathsf{WMSO}{+}\mathsf{B})$-formula α back into a $\mathsf{Bool}(\mathsf{MSO},\mathsf{WMSO}{+}\mathsf{B})$-formula α' such that $(\mathcal{G}_\mathcal{T} \models \alpha \Leftrightarrow \mathcal{T} \models \alpha')$, then we can finish the proof.

Recall the construction of $\mathcal{G}_\mathcal{T}$: For every node $v \in D$ of $\mathcal{T} = (D,\to,\rho)$ we introduce $m := |\mathsf{V}_\varphi|$ copies (v,x) for $x \in \mathsf{V}_\varphi$. The $\mathcal{S}$-relations between these nodes are determined by the propositions $p_1,\ldots,p_n$: The interpretation of $r \in \mathcal{S}$ contains all k-tuples $(k = \mathsf{ar}(r))$ $((su_1,y_1),\ldots,(su_k,y_k))$ for which there exist $1 \le i \le n$ and $u \in [1,d]^*$ with $|u| = d_i$, $p_i \in \rho(su)$, $R_i = r(\mathsf{X}^{j_1}y_1,\ldots,\mathsf{X}^{j_k}y_k)$, and $u_t = u[:j_t]$ for $1 \le t \le k$. This is a particular case of an MSO-transduction [7] with copy number m. It is therefore possible to compute from a given MSO-sentence η over the signature $\mathcal{S}$ an MSO-sentence η' such that $\mathcal{G}_\mathcal{T} \models \eta \Leftrightarrow \mathcal{T} \models \eta'$. But the problem is that in our situation η is the $\mathsf{Bool}(\mathsf{MSO},\mathsf{WMSO}{+}\mathsf{B})$-formula α, and it is not clear whether MSO-transductions (or even first-order interpretations) are compatible with the logic $\mathsf{WMSO}{+}\mathsf{B}$. Nevertheless, there is a simple solution. Let $\mathsf{V}_\varphi = \{x_1,\ldots,x_m\}$. From a Kripke d-tree $\mathcal{T} = ([1,d]^*,\to,\rho)$ we build an extended $(d+m)$-Kripke tree $\mathcal{T}^e = ([1,d+m]^*,\to,\rho^e)$ as follows: Let us fix new propositions $q_1,\ldots,q_m$ (one for each variable x_i) that do not occur in the MSO-sentence ψ and such that $\rho(v) \cap \{q_1,\ldots,q_m\} = \emptyset$ for all $v \in [1,d]^*$. We define the new labeling function ρ^e as follows:

$$\begin{aligned}
\rho^e(v) &= \rho(v) \text{ for } v \in [1,d]^* \\
\rho^e(vi) &= \{q_{i-d}\} \text{ for } v \in [1,d]^*, d+1 \le i \le d+m \\
\rho^e(viu) &= \emptyset \text{ for } v \in [1,d]^*, d+1 \le i \le d+m, u \in [1,d+m]^+
\end{aligned}$$

It is easy to write down an MSO-sentence β such that for every $(d+m)$-Kripke tree $\mathcal{T}'$ we have $\mathcal{T}' \models \beta$ if and only if $\mathcal{T}' \cong \mathcal{T}^e$ for some Kripke d-tree $\mathcal{T}$. Moreover, since the old Kripke d-tree $\mathcal{T}$ is MSO-definable within $\mathcal{T}^e$, we can construct from the MSO-sentence ψ a new MSO-sentence ψ^e such that $\mathcal{T} \models \psi$ if and only if $\mathcal{T}^e \models \psi^e$. Finally, let $q(x) = \bigvee_{i=1}^m q_i(x)$. Then, the nodes of $\mathcal{G}_\mathcal{T}$ are in a natural bijection with

the nodes of $\mathcal{T}^e$ that satisfy $q(x)$: If $\mathcal{T}^e \models q(u)$ for $u \in [1, d+m]^*$, then there is a unique $i \in [1, m]$ such that $\mathcal{T}^e \models q_i(u)$ and $u = v(i+d)$. Then we associate the node u with node (v, x_i) of $\mathcal{G}_\mathcal{T}$. By relativizing all quantifiers in the $\mathsf{Bool}(\mathsf{MSO}, \mathsf{WMSO}{+}\mathsf{B})$-formula α to $q(x)$, we can construct a $\mathsf{Bool}(\mathsf{MSO}, \mathsf{WMSO}{+}\mathsf{B})$-formula α^e such that $\mathcal{G}_\mathcal{T} \models \alpha$ if and only if $\mathcal{T}^e \models \alpha^e$.

It follows that there is a Kripke d-tree $\mathcal{T}$ such that $\mathcal{T} \models \psi$ and $\mathcal{G}_\mathcal{T} \models \alpha$ if and only if there is a Kripke $(d+m)$-tree $\mathcal{T}'$ such that $\mathcal{T}' \models (\beta \wedge \psi^e \wedge \alpha^e)$. Since $\beta \wedge \psi^e \wedge \alpha^e$ is a $\mathsf{Bool}(\mathsf{MSO}, \mathsf{WMSO}{+}\mathsf{B})$-formula, the latter is decidable by Thm. 3.

6 Concrete Domains over the Integers

The main technical result of this section is:

Proposition 13. *$\mathcal{Z}$ from* (1) *has the property* $\mathsf{EHomDef}(\mathsf{Bool}(\mathsf{MSO}, \mathsf{WMSO}{+}\mathsf{B}))$.

Since $\mathcal{Z}$ is negation-closed (see Ex. 1) our main result follows by Thm. 9:

Theorem 14. $\mathsf{SATCTL}^*(\mathcal{Z})$ *is decidable.*

We prove Prop. 13 in three steps. First, we show that the structure $(\mathbb{Z}, <)$ has the property $\mathsf{EHomDef}(\mathsf{WMSO}{+}\mathsf{B})$. Then we extend this result to the structure $(\mathbb{Z}, <, =)$ and, finally, to the full structure $\mathcal{Z}$.

Proposition 15. $(\mathbb{Z}, <)$ *has the property* $\mathsf{EHomDef}(\mathsf{WMSO}{+}\mathsf{B})$.

As a preparation of the proof, we first define some terminology and then we characterize structures that allow homomorphisms to $(\mathbb{Z}, <)$ in terms of their paths. Let $\mathcal{A} = (A, I)$ be a countable $\{<\}$-structure. We identify $\mathcal{A}$ with the directed graph (A, E) where $E = I(<)$. When talking about paths, we always refer to finite directed E-paths. The length of a path $(a_0, a_1, \ldots, a_n)$ (i.e., $(a_{i-1}, a_i) \in E$ for $1 \leq i \leq n$) is n. For $S \subseteq A$ and $x \in A \setminus S$, a path from x to S is a path from x to some node $y \in S$. A path from S to x is defined in a symmetric way.

Lemma 16. *We have $\mathcal{A} \preceq (\mathbb{Z}, <)$ if and only if*

(H1) $\mathcal{A}$ does not contain cycles, and
(H2) for all $a, b \in A$ there is $c \in \mathbb{N}$ such that the length of all paths from a to b is bounded by c.

Proof. Let us first show the "only if" direction of the lemma. Suppose h is a homomorphism from $\mathcal{A}$ to $(\mathbb{Z}, <)$. The presence of a cycle $(a_0, \ldots a_{k-1})$ in $\mathcal{A}$ ($k \geq 1$, $(a_i, a_{i+1 \bmod k}) \in E$ for $0 \leq i \leq k-1$) would imply the existence of integers $z_0, \ldots z_{k-1}$ with $z_i < z_{i+1 \bmod k}$ for $0 \leq i \leq k-1$ (where $z_i = h(a_i)$), which is not possible. Hence, (H1) holds.

Suppose now that $a, b \in A$ are such that for every n there is a path of length at least n from a to b. If $d = h(b) - h(a)$, we can find a path $(a_0, a_1 \ldots, a_k)$ with $a_0 = a$, $a_k = b$ and $k > d$. Since h is a homomorphism, this path will be mapped to an increasing sequence of integers $h(a) = h(a_0) < h(a_1) < \cdots < h(a_k) = h(b)$. But this contradicts $h(b) - h(a) = d < k$. Hence, (H2) holds.

For the "if" direction of the lemma assume that $\mathcal{A}$ is acyclic (property (H1)) and that (H2) holds. Fix an enumeration $a_0, a_1, a_2, \ldots$ of the countable set A. For $n \geq 0$ let $S_n := \{a \in A \mid \exists i, j \leq n : (a_i, a), (a, a_j) \in E^*\}$, which has the following properties:

(P1) S_n is convex w.r.t. the partial order E^*: If $a, c \in S_n$ and $(a, b), (b, c) \in E^*$, then $b \in S_n$.
(P2) For $a \in A \setminus S_n$ all paths between a and S_n are "one-way", i.e., there do not exist $b, c \in S_n$ such that $(b, a), (a, c) \in E^*$. This follows from (P1).
(P3) For all $a \in A \setminus S_n$ there exists a bound $c \in \mathbb{N}$ such that all paths between a and S_n have length at most c. Let $c_n^a \in \mathbb{N}$ be the smallest such bound (hence, we have $c_n^a = 0$ if there do not exist paths between a and S_n).

To see (P3), assume that there only exist paths from S_n to a but not the other way round (see (P2)); the other case is symmetric. If there is no bound on the length of paths from S_n to a, then by definition of S_n, there is no bound on the length of paths from $\{a_0, \ldots, a_n\}$ to a. By the pigeon principle, there exists $0 \leq i \leq n$ such that there is no bound on the length of paths from a_i to a. But this contradicts property (H2).

We build our homomorphism h inductively. For every $n \geq 0$ we define functions $h_n : S_n \to \mathbb{Z}$ such that the following invariants hold for all $n \geq 0$.

(I1) If $n > 0$ then $h_n(a) = h_{n-1}(a)$ for all $a \in S_{n-1}$
(I2) $h_n(S_n)$ is bounded in $\mathbb{Z}$, i.e., there exist $z_1, z_2 \in \mathbb{Z}$ such that $h_n(S_n) \subseteq [z_1, z_2]$.
(I3) h_n is a homomorphism from the subgraph $(S_n, E \cap (S_n \times S_n))$ to $(\mathbb{Z}, <)$.

For $n = 0$ we have $S_0 = \{a_0\}$. We set $h_0(a_0) = 0$ (any other integer would be also fine). Properties (I1)–(I3) are easily verified. For $n > 0$, there are four cases.

Case 1. $a_n \in S_{n-1}$, thus $S_n = S_{n-1}$. We set $h_n = h_{n-1}$. Clearly, (I1)–(I3) hold for n.

Case 2. $a_n \notin S_{n-1}$ and there is no path from a_n to S_{n-1} or vice versa. We set $h_n(a_n) := 0$ (and $S_n = S_{n-1} \cup \{a_n\}$). In this case (I1)–(I3) follow easily from the induction hypothesis.

Case 3. $a_n \notin S_{n-1}$ and there exist paths from a_n to S_{n-1}. Then, by (P2) there do not exist paths from S_{n-1} to a_n. Hence, we have

$$S_n = S_{n-1} \cup \{a \in A \mid \exists b \in S_{n-1} : (a_n, a), (a, b) \in E^*\}.$$

We have to assign a value $h_n(a)$ for all $a \in A \setminus S_{n-1}$ that lie along a path from a_n to S_{n-1}. By (I2) there exist $z_1, z_2 \in \mathbb{Z}$ with $h_{n-1}(S_{n-1}) \subseteq [z_1, z_2]$. Recall the definition of c_{n-1}^a from (P3). For all $a \in A \setminus S_{n-1}$ that lie on a path from a_n to S_{n-1}, we set $h_n(a) := z_1 - c_{n-1}^a$. Since there are paths from a to S_{n-1}, we have $c_{n-1}^a > 0$. Hence, for all $a \in S_n \setminus S_{n-1}$, $h_n(a) < z_1$. Let us check that $h_n : S_n \to \mathbb{Z}$ satisfy (I1)– (I3): Invariant (I1) holds by definition of h_n. For (I2) note that $h_n(S_n) \subseteq [z_1 - c_{n-1}^{a_n}, z_2]$.

It remains to show (I3), i.e., that h_n is a homomorphism from $(S_n, E \cap (S_n \times S_n))$ to $(\mathbb{Z}, <)$. Hence, we have to show that $h(b_1) < h(b_2)$ for all $(b_1, b_2) \in E \cap (S_n \times S_n)$.

- If $b_1, b_2 \in S_{n-1}$, then $h_n(b_1) = h_{n-1}(b_1) < h_{n-1}(b_2) = h_n(b_2)$ by induction hypothesis.
- If $b_1 \in S_n \setminus S_{n-1}$ and $b_2 \in S_{n-1}$, we know that $h_n(b_2) = h_{n-1}(b_2) \geq z_1$ while $h_n(b_1) < z_1$ by construction. This directly implies $h_n(b_1) < h_n(b_2)$.

- If $b_2 \in S_n \setminus S_{n-1}$ and $b_1 \in S_{n-1}$, then $(b_1, b_2) \in E$ and by assumption b_2 must be on a path from a_n to S_{n-1} which contradicts (P2).
- If both b_1 and b_2 belong to $S_n \setminus S_{n-1}$ then $h_n(b_i) := z_1 - c_{n-1}^{b_i}$ for $i \in \{1, 2\}$ Since $(b_1, b_2) \in E$, we have $c_{n-1}^{b_1} > c_{n-1}^{b_2}$. This implies $h_n(b_1) < h_n(b_2)$.

Case 4. $a_n \notin S_{n-1}$ and there exist paths from S_{n-1} to a_n. For all $a \in S_n \setminus S_{n-1} = \{a \in A \setminus S_{n-1} \mid a \text{ belongs to a path from } S_{n-1} \text{ to } a_n\}$, set $h_n(a) = z_2 + c_{n-1}^a$. The rest of the argument goes analogously to Case 3.

This concludes the construction of h_n. By (I1) limit function $h = \bigcup_{i \in \mathbb{N}} h_i$ exists. By (I3) and $A = \bigcup_{i \in \mathbb{N}} S_i$, h is a homomorphism from $\mathcal{A}$ to $(\mathbb{Z}, <)$. □

Proof of Prop. 15. We translate the conditions (H1) and (H2) from Lemma 16 into WMSO+B. Cycles are excluded by the sentence $\neg \mathsf{ECycle}_<$ (Example 2). Moreover, for an acyclic $\{<\}$-structure $\mathcal{A}$ we have $\mathcal{A} \models \forall x \forall y\ \mathsf{BPaths}_<(x, y)$ (see also Example 2) if and only if for all $a, b \in A$ there is a bound $b \in \mathbb{N}$ on the length of paths from a to b. Thus, $\mathcal{A} \preceq (\mathbb{Z}, <)$ if and only if $\mathcal{A} \models \neg \mathsf{ECycle}_< \wedge \forall x \forall y\ \mathsf{BPaths}_<(x, y)$. □

Next, we extend Prop. 15 to the negation-closed structure $(\mathbb{Z}, <, =)$. To do so let us fix a countable $\{<, =\}$-structure $\mathcal{A} = (A, I)$. Note that $I(=)$ is not necessarily the identity relation on A. Let $\sim = (I(=) \cup I(=)^{-1})^*$ be the smallest equivalence relation on A that contains $I(=)$. Since $\sim$ is the reflexive and transitive closure of the first-order definable relation $I(=) \cup I(=)^{-1}$, we can construct a WMSO-formula $\tilde{\varphi}(x, y)$ (using the reach-construction from Ex. 2) that defines $\sim$. Let

$$E_< = \sim \circ\, I(<) \circ \sim \text{ i.e., the relation defined by the formula} \tag{2}$$

$$\varphi_<(x, y) = \exists u\ \exists v\ (\tilde{\varphi}(x, u) \wedge u < v \wedge \tilde{\varphi}(v, y)). \tag{3}$$

With $\tilde{\mathcal{A}} = (\tilde{A}, \tilde{I})$ we denote the $\sim$-quotient of $\mathcal{A}$: It is a $\{<\}$-structure, its domain is the set $\tilde{A} = \{[a]_\sim \mid a \in A\}$ of all $\sim$-equivalence classes, and for two equivalence classes $[a]_\sim$ and $[b]_\sim$ we have $([a]_\sim, [b]_\sim) \in \tilde{I}(<)$ iff there are $a' \sim a$ and $b' \sim b$ such that $(a', b') \in I(<)$. Let us write $[a]$ for $[a]_\sim$. We have:

Lemma 17. *$\mathcal{A} \preceq (\mathbb{Z}, <, =)$ if and only $\tilde{\mathcal{A}} \preceq (\mathbb{Z}, <)$.*

In the next lemma, we translate the conditions for the existence of a homomorphism from $\tilde{\mathcal{A}}$ to $(\mathbb{Z}, <)$ into conditions in terms of $\mathcal{A}$.

Lemma 18. *The following conditions are equivalent:*

- *$\tilde{\mathcal{A}}$ satisfies the conditions (H1) and (H2) from Lemma 16.*
- *The graph $(A, E_<)$ is acyclic and for all $a, b \in A$ there is a bound $c \in \mathbb{N}$ such that all $E_<$-paths from a to b have length at most c.*

Proposition 19. *The concrete domains $(\mathbb{Z}, <, =)$, $(\mathbb{N}, <, =)$ and $(\mathbb{Z} \setminus \mathbb{N}, <, =)$ have property* EHomDef(WMSO+B).

Proof. We only proof the proposition for $(\mathbb{Z}, <, =)$. The other two cases are similar. We want to find a (WMSO+B)-formula φ such that for all $\{<, =\}$-structures $\mathcal{A}$, $\mathcal{A} \models \varphi$ if and only if $\mathcal{A} \preceq (\mathbb{Z}, <, =)$. Let $\mathcal{A} = (A, I)$ be a $\{<, =\}$-structure. We use the

notations introduced before Lemma 17. By Lemma 17 and 18 we have to construct a (WMSO+B)-formula expressing that $\mathcal{A}$ has no $E_<$-cycles and for all $a, b \in A$ there is a bound $c \in \mathbb{N}$ on the length of $E_<$-paths from a to b. For this, we can use the formula constructed in the proof of Prop. 15 with $<$ replaced by the formula $\varphi_<$ from (3). □

In the rest of this section, we prove Prop. 19 for the full structure $\mathcal{Z}$ from (1), which is defined over the infinite signature $\mathcal{S} = \{<, =\} \cup \{=_c |\ c \in \mathbb{Z}\} \cup \{\equiv_{a,b} |\ 0 \le a < b\}$. By the definition of $\mathsf{EHomDef}(\mathsf{Bool}(\mathsf{MSO}, \mathsf{WMSO}+\mathsf{B}))$ we have to compute from a finite subsignature $\sigma \subseteq \mathcal{S}$ a $\mathsf{Bool}(\mathsf{MSO}, \mathsf{WMSO}+\mathsf{B})$-sentence φ_σ that defines the existence of a homomorphism to $\mathcal{Z}$ when interpreted over a σ-structure $\mathcal{A}$. Hence, let us fix a finite subsignature $\sigma \subseteq \mathcal{S}$. We can assume that $\sigma = \{<, =\} \cup \{=_c |\ c \in C\} \cup \{\equiv_{a,b} |\ b \in D, 0 \le a < b\}$ for finite non-empty sets $C \subseteq \mathbb{Z}$ and $D \subseteq \mathbb{N} \setminus \{0, 1\}$. Define $m = \min(C)$ and $M = \max(C)$. W.l.o.g. we can assume that $m \le 0$ and $M \ge 0$. Let $\mathcal{A} = (A, I)$ be a countable σ-structure. In order to not confuse the relation $I(=)$ with the identity relation on A, we write in the following $E_=(x, y)$ for the atomic formula expressing that (x, y) belongs to the relation $I(=)$. Similarly, we write $E_c(x)$ for the atomic formula expressing that $x \in I(=_c)$. Instead of $\equiv_{a,b}(x)$ we write $x \equiv a \bmod b$.

Define $x \le y \Leftrightarrow (x < y \vee E_=(x, y) \vee E_=(y, x))$ and the MSO-formula

$$\varphi_{\text{bounded}}(x) = \exists y\, \exists z \big(\bigvee_{c \in C} E_c(y) \wedge \bigvee_{c \in C} E_c(z) \wedge \mathsf{reach}_{\le}(y, x) \wedge \mathsf{reach}_{\le}(x, z) \big).$$

Let $B = \{a \in A \mid \mathcal{A} \models \varphi_{\text{bounded}}(a)\}$. We call the induced substructure $\mathcal{B} := \mathcal{A}\restriction_B$ the "bounded" part of $\mathcal{A}$. Every homomorphism from $\mathcal{B}$ to $\mathcal{Z}$ has to map B to the interval $[m, M]$. Thus, a homomorphism $h : \mathcal{B} \to \mathcal{Z}$ can be identified with a partition of B into $M - m + 1$ sets $B_m, \ldots, B_M$, where $B_i = \{a \in B \mid h(a) = i\}$. It follows that:

Lemma 20. *There is an* MSO*-sentence φ_B such that for every $\mathcal{S}$-structure $\mathcal{A}$ with bounded part $\mathcal{B}$, we have $\mathcal{B} \preceq \mathcal{Z}$ if and only if $\mathcal{A} \models \varphi_B$.*

Similar to B we define three other parts of a σ-structure by the WMSO-formulas

$$\begin{aligned}
\varphi_{\text{greater}}(x) &= \neg\varphi_{\text{bounded}}(x) \wedge \exists y \, \big(\varphi_{\text{bounded}}(y) \wedge \mathsf{reach}_{\le}(y, x)\big), \\
\varphi_{\text{smaller}}(x) &= \neg\varphi_{\text{bounded}}(x) \wedge \exists y \, \big(\varphi_{\text{bounded}}(y) \wedge \mathsf{reach}_{\le}(x, y)\big), \\
\varphi_{\text{rest}}(x) &= \neg(\varphi_{\text{bounded}}(x) \vee \varphi_{\text{greater}}(x) \vee \varphi_{\text{smaller}}(x)).
\end{aligned}$$

Moreover, let $G = \{a \in A \mid \mathcal{A} \models \varphi_{\text{greater}}(a)\}$, $S = \{a \in A \mid \mathcal{A} \models \varphi_{\text{smaller}}(a)\}$, and $R = \{a \in A \mid \mathcal{A} \models \varphi_{\text{rest}}(a)\}$. Let $\mathcal{N} = \mathcal{Z}\restriction_{\mathbb{N}}$ and $\overline{\mathcal{N}} = \mathcal{Z}\restriction_{\mathbb{Z}\setminus\mathbb{N}}$. Then we have:

Lemma 21. $\mathcal{A} \preceq \mathcal{Z}$ *iff* $\big(\mathcal{B} \preceq \mathcal{Z}, \mathcal{A}\restriction_{G \cup S \cup R} \preceq \mathcal{Z}, \mathcal{A}\restriction_G \preceq \mathcal{N},$ *and* $\mathcal{A}\restriction_S \preceq \overline{\mathcal{N}}\big)$.

We need some conventions on modulo constraints. A sequence $(a_1, b_1), \ldots, (a_k, b_k)$ with $0 \le a_i < b_i \in D$ for $1 \le i \le k$ is *contradictory*, if there is no number $n \in \mathbb{N}$ such that $n \equiv a_i \bmod b_i$ for all $1 \le i \le k$. In the following let CS_k denote the set of contradictory sequences of length k. It is straightforward to show that every contradictory sequence contains a contradictory subsequence of length at most $\ell := \max\{2, |D|\}$.

Recall that $\sim$ is the smallest equivalence relation containing $I(=)$ and that $\sim$ is defined by the WMSO-formula $\tilde{\varphi}(x, y)$. We call a σ-structure $\mathcal{A} = (A, I)$ *modulo*

contradicting if there is a $\sim$-class $[c]$, elements $c_1, c_2, \ldots, c_k \in [c]$, and a contradictory sequence $(a_1, b_1), \ldots, (a_k, b_k)$ such that $c_i \in I(\equiv_{a_i,b_i})$ for all $1 \leq i \leq k$.

The following WMSO-formula φ_{modcon} expresses that a σ-structure is modulo contradicting, where we write $s_a(j)$ (resp. $s_b(j)$) for the first (resp. second) entry of the j-th element of the sequence $s \in \mathsf{CS}_k$:

$$\varphi_{\text{modcon}} = \bigvee_{2 \leq k \leq \ell} \bigvee_{s \in \mathsf{CS}_k} \exists x_1 \cdots \exists x_k \bigwedge_{i,j \leq k} \tilde{\varphi}(x_i, x_j) \wedge \bigwedge_{j \leq k} x_j \equiv s_a(j) \bmod s_b(j)$$

Lemma 22. *Let $\sigma' = \sigma \setminus \{=_c \mid c \in \mathbb{Z}\}$. Let $\mathcal{A} = (A, I)$ be a σ'-structure.*

- *$\mathcal{A} \preceq \mathcal{Z}$ iff $\mathcal{A}$ is not modulo contradicting and $(A, I(<), I(=)) \preceq (\mathbb{Z}, <, =)$.*
- *$\mathcal{A} \preceq \mathcal{N}$ iff $\mathcal{A}$ is not modulo contradicting and $(A, I(<), I(=)) \preceq (\mathbb{N}, <, =)$.*

Proof of Prop. 13. Let $\mathcal{A} = (A, I)$ be a σ-structure. We defined a partition of A into B, G, S, and R. Since membership in each of these sets is $(\mathsf{WMSO}+\mathsf{B})$-definable, we can relativize any $(\mathsf{WMSO}+\mathsf{B})$-formula to any of these sets. For instance, we write φ^G for the relativization of φ to the substructure induced by G. Let φ_B be the MSO-formula from Lemma 20, and for $C \in \{\mathbb{Z}, \mathbb{N}, \mathbb{Z} \setminus \mathbb{N}\}$ let φ_C be a formula that expresses $\mathcal{A} \preceq (C, <, =)$, see Prop. 19. Then $\mathcal{A} \models (\varphi_B \wedge \varphi_{\mathbb{Z}}^{G \cup S \cup R} \wedge \varphi_{\mathbb{N}}^{G} \wedge \varphi_{\mathbb{Z} \setminus \mathbb{N}}^{S} \wedge \neg \varphi_{\text{modcon}})$ iff $\mathcal{A} \preceq \mathcal{Z}$ due to Lemmas 21 and 22. □

7 Extensions, Applications, Open Problems

A simple adaptation of our proof for $\mathcal{Z}$ shows that $\mathcal{Q} = (\mathbb{Q}, <, =, (=_q)_{q \in \mathbb{Q}})$ has the property $\mathsf{EHomDef}(\mathsf{Bool}(\mathsf{MSO}, \mathsf{WMSO}+\mathsf{B}))$ as well: $\mathcal{A} = (A, I) \preceq \mathcal{Q}$ iff (i) $(A, E_<)$ is acyclic, where $E_<$ is defined as in (2), (ii) there does not exist $(a, b) \in E_<^+$ (the transitive closure of $E_<$) with $a \in I(=_p)$, $b \in I(=_q)$ and $q \leq p$, and (iii) there do not exist $a \sim b$ with $a \in I(=_p)$, $b \in I(=_q)$, and $q \neq p$.

Let us finally state a simple preservation theorem for $\mathcal{A}$-satisfiability for $\mathsf{CTL}^*(\mathcal{S})$. Assume that $\mathcal{A}$ and $\mathcal{B}$ are structures over countable signatures $\mathcal{S}_\mathcal{A}$ and $\mathcal{S}_\mathcal{B}$, respectively, and let B be the domain of $\mathcal{B}$. We say that $\mathcal{A}$ is *existentially interpretable* in $\mathcal{B}$ if there exist $n \geq 1$ and quantifier-free first-order formulas $\varphi(y_1, \ldots, y_l, x_1, \ldots, x_n)$ and

$$\varphi_r(z_1, \ldots, z_{l_r}, x_{1,1}, \ldots, x_{1,n}, \ldots, x_{\mathsf{ar}(r),1}, \ldots, x_{\mathsf{ar}(r),n}) \text{ for } r \in \mathcal{S}_\mathcal{A}$$

over the signature $\mathcal{S}_\mathcal{B}$, where the mapping $r \mapsto \varphi_r$ has to be computable, such that $\mathcal{A}$ is isomorphic to the structure $(\{\overline{b} \in B^n \mid \exists \overline{c} \in B^l \colon \mathcal{B} \models \varphi(\overline{c}, \overline{b})\}, I)$ with

$$I(r) = \{(\overline{b}_1, \ldots, \overline{b}_{\mathsf{ar}(r)}) \in B^{\mathsf{ar}(r)n} \mid \exists \overline{c} \in B^{l_r} \colon \mathcal{B} \models \varphi_r(\overline{c}, \overline{b}_1, \ldots, \overline{b}_{\mathsf{ar}(r)})\} \text{ for } r \in \mathcal{S}_\mathcal{A}.$$

Proposition 23. *If $\mathsf{SATCTL}^*(\mathcal{B})$ is decidable and $\mathcal{A}$ is existentially interpretable in $\mathcal{B}$, then $\mathsf{SATCTL}^*(\mathcal{A})$ is decidable too.*

Examples of structures $\mathcal{A}$ that are existentially interpretable in $(\mathbb{Z}, <, =)$, and hence have a decidable $\mathsf{SATCTL}^*(\mathcal{A})$-problem are (i) $(\mathbb{Z}^n, <_{\text{lex}}, =)$ (for $n \geq 1$), where $<_{\text{lex}}$ denotes the strict lexicographic order on n-tuples of integers, and (ii) the structure

$\text{Allen}_{\mathbb{Z}}$, which consists of all $\mathbb{Z}$-intervals together with Allen's relations b (before), a (after), m (meets), *mi* (met-by), o (overlaps), *oi* (overlapped by), d (during), *di* (contains), s (starts), *si* (started by), f (ends), *fi* (ended by). In artificial intelligence, Allen's relations are a popular tool for representing temporal knowledge.

It remains open to determine the complexity of CTL^*-satisfiability with constraints over $\mathcal{Z}$, see the last paragraph in the introduction. Clearly, this problem is 2EXPTIME-hard due to the known lower bound for CTL^*-satisfiability. To get an upper complexity bound, one should investigate the complexity of the emptiness problem for puzzles from [1] (see Lemma 5). An interesting structure for which the decidability status for satisfiability of CTL^* with constraints is open, is $(\{0,1\}^*, \leq_p, \not\leq_p)$, where $\leq_p$ is the prefix order on words, and $\not\leq_p$ is its complement. It is not clear, whether this structure has the property $\mathsf{EHomDef}(\mathsf{Bool}(\mathsf{MSO}, \mathsf{WMSO}+\mathsf{B}))$.

Acknowledgments. We are grateful to Szymon Toruńczyk for fruitful discussions.

References

1. Bojańczyk, M., Toruńczyk, S.: Weak MSO+U over infinite trees. In: Proc. STACS 2012. LIPIcs, vol. 14, pp. 648–660. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2012)
2. Bojańczyk, M., Toruńczyk, S.: Weak MSO+U over infinite trees (long version), `http://www.mimuw.edu.pl/~bojan/papers/wmsou-trees.pdf`
3. Bozzelli, L., Gascon, R.: Branching-time temporal logic extended with qualitative Presburger constraints. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 197–211. Springer, Heidelberg (2006)
4. Carapelle, C., Kartzow, A., Lohrey, M.: Satisfiability of CTL* with constraints Technical report, arXiv.org (2013), `http://arxiv.org/abs/1306.0814`
5. Čerāns, K.: Deciding properties of integral relational automata. In: Shamir, E., Abiteboul, S. (eds.) ICALP 1994. LNCS, vol. 820, pp. 35–46. Springer, Heidelberg (1994)
6. Colcombet, T., Löding, C.: Regular cost functions over finite trees. In: Proc. LICS 2010, pp. 70–79. IEEE Computer Society (2010)
7. Courcelle, B.: The monadic second-order logic of graphs V: On closing the gap between definability and recognizability. Theor. Comput. Sci. 80(2), 153–202 (1991)
8. Demri, S., D'Souza, D.: An automata-theoretic approach to constraint LTL. Inf. Comput. 205(3), 380–415 (2007)
9. Demri, S., Gascon, R.: Verification of qualitative $\mathbb{Z}$ constraints. Theor. Comput. Sci. 409(1), 24–40 (2008)
10. Gascon, R.: An automata-based approach for CTL* with constraints. Electr. Notes Theor. Comput. Sci. 239, 193–211 (2009)
11. Lutz, C.: Description logics with concrete domains-a survey. In: Advances in Modal Logic 4, pp. 265–296. King's College Publications (2003)
12. Lutz, C.: Combining interval-based temporal reasoning with general TBoxes. Artificial Intelligence 152(2), 235–274 (2004)
13. Lutz, C.: NEXPTIME-complete description logics with concrete domains. ACM Trans. Comput. Log. 5(4), 669–705 (2004)
14. Lutz, C., Milicic, M.: A tableau algorithm for description logics with concrete domains and general TBoxes. J. Autom. Reasoning 38(1-3), 227–259 (2007)

Proof Graphs for Parameterised Boolean Equation Systems

Sjoerd Cranen, Bas Luttik, and Tim A.C. Willemse

Eindhoven University of Technology

Abstract. Parameterised Boolean equation systems (PBESs) can be used for solving a variety of verification problems such as model checking and equivalence checking problems. The definition of solution for a PBES is notoriously difficult to understand, which makes them hard to work with. Tan and Cleaveland proposed a notion of proof for Boolean equation systems they call *support sets*. We show that an adapted notion of support sets called *proof graphs* gives an alternative characterisation of the solution to a PBES, and prove that minimising proof graphs is NP-hard. Finally, we explain how proof graphs may be used in practice and illustrate how they can be used in equivalence checking to generate distinguishing formulas.

1 Introduction

Boolean equation systems (BESs) are the enabling technology in tool suites such as CADP, the Concurrency Workbench NC and the Edinburgh Concurrency Workbench, for analysing complex, concurrent systems. For instance, they are used to encode model checking problems for the modal μ-calculus and for deciding a variety of process equivalences. *Parameterised* Boolean equation systems (PBESs) [9], used in the tool suite mCRL2, extend BESs with data and first-order quantification, thereby lifting the typical finiteness restrictions of BESs. Consequently, PBESs can be used to encode equivalence checking problems for infinitely large and infinitely branching systems, but also for encoding model checking problems for μ-calculi with first-order and real-time extensions. A variety of techniques exist for solving a PBES; using pattern recognition, they can sometimes be solved by simply looking up a solution to the pattern; using abstraction techniques (that are again defined completely within the domain of parameterised Boolean equation systems), PBESs with infinite data domains can in some cases be reduced to PBESs with finite data domains; using instantiation, they can be reduced to Boolean equation systems.

Relying on PBES technology for verification has major benefits: solving algorithms for PBESs and all solution-preserving transformations available to PBESs instantly become available to all decision problems that are encoded as PBESs. Moreover, the clear-cut separation between the encoding of a verification problem as a PBES, and the technology for solving PBESs leads to improved maintenance and flexibility. The close link with Boolean equation systems and parity games allows exploration and solving techniques from these contexts to be re-used, and the combination of many-sorted logic and recursion shows great similarity with process algebra.

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 470–484, 2013.

Tools that implement solving algorithms for PBESs typically produce an end result that is a simple *true* or *false* verdict. Understanding the correctness of the outcome is hardly helped by such verdicts. In the case of an unexpected outcome, or in case the answer is as expected but the person carrying out the verification would like to ascertain that it is the answer to the right question, additional information is needed to explain the answer. Explaining the answer in terms of the formal definition of the solution is often quite involved and not very illuminating. Nevertheless, a more structural way of extracting a meaningful explanation or some form of diagnostics from a solution to a PBES is currently lacking.

We propose a notion of proof called *proof graphs*, that solve exactly these issues: they offer a formal, intuitive explanation of the solution of a PBES, and they are sufficiently detailed to serve as the basis for presenting diagnostics. Our proof graphs are inspired by Tan and Cleaveland's *support sets* [13]. Support sets were studied in the setting of *closed* BESs in standard recursive form, a fragment of BESs in which mixing of $\wedge$ and $\vee$ in right-hand sides of an equation is prohibited.

We lift these restrictions by defining proof graphs on arbitrary PBESs: they need not be closed, nor need they be in standard recursive form. The latter is practical, because transformations to this form may cause an exponential blow-up in the size of the PBES or yield a PBES that cannot be instantiated with the current techniques. Any such transformation changes the structure of the PBES, making it difficult to relate a proof graph for the transformed PBES back to the original PBES.

Section 2 briefly reviews the concept of parameterised Boolean equation systems, after which we introduce proof graphs in Section 3. We show that the existence of a proof graph for a variable instantiation in a PBES is sufficient to determine its solution in 3.1, and in Section 3.2 we show that for every solution to a PBES, there is a proof graph that explains it. We note that our less restrictive setting required different proof strategies than Tan and Cleaveland used for support sets. For instance, the completeness proof for support sets relies on a correspondence between the syntax of BESs in standard recursive form and the structure of support sets. In the presence of first-order constructs and potentially infinite semantic domains in a PBES, no such correspondence can be exploited.

Section 3.3 explores the concept of minimality for proof graphs, and shows that minimising proof graphs is NP-hard in general, and NP-complete if the data language is decidable in polynomial time. We finish with an outlook on the practical implications of proof graphs and illustrate some of these through examples in Section 4.

2 Preliminaries

In this paper we use the notion of *parameterised Boolean equation system* (PBES) as defined in [9]. In the remainder we will often simply call them 'equation systems'.

We assume the use of a theory of abstract data types that gives rise to *data terms*. For the sake of readability, we assume that there is only one data sort D with semantic domain $\mathbb{D}$ in our proofs. In practice, various sorts D_i with semantic domains $\mathbb{D}_i$ may be used, but it is easy to see this is equivalent to using projection functions on a semantic domain $\mathbb{D}$ that contains all elements of every $\mathbb{D}_i$. In our examples, we assume the

existence of data sorts B and N representing the Booleans $\mathbb{B} = \{\mathbb{t}, \mathbb{f}\}$ and the natural numbers $\mathbb{N}$. For these types, we do not distinguish between the syntactic elements from B and N, and the semantic elements from $\mathbb{B}$ and $\mathbb{N}$ they represent.

Data terms may contain unbound variables from some set of variable names $\mathcal{V}$. A *data environment* $\delta: \mathcal{V} \to \mathbb{D}$ maps variable names to their values. We assume the existence of a mapping $[\![\_]\!]\delta$ for the data language that is used which maps expressions of sort D to their semantic value in $\mathbb{D}$, using a data environment δ to resolve open variables. We will assume $\mathbb{B} \subseteq \mathbb{D}$, so that we may define that b is a Boolean data term if and only if $[\![b]\!]\delta \in \mathbb{B}$. We use square brackets to denote environment updates: $\delta[v/d]$ is the environment that maps d to v, and maps everything else in the same way δ does.

Definition 1 (PBES Syntax). *We denote the* empty *equation system by* $\emptyset$. Equation systems $\mathcal{E}$ *and* predicate formulas ϕ *are defined through the following grammar:*

$$\mathcal{E} ::= \emptyset \mid (\mu X(d{:}\, D) = \phi)\, \mathcal{E} \mid (\nu X(d{:}\, D) = \psi)\, \mathcal{E}$$
$$\phi, \psi ::= b \mid \phi \wedge \psi \mid \phi \vee \psi \mid \forall_{d:\, D}\, \phi \mid \exists_{d:\, D}\, \phi \mid X(e)$$

Here, b *is a Boolean data term,* e *a data expression,* X *a predicate variable in some presupposed set* $\mathcal{P}$ *of predicate variables, and* d *a data variable of sort* D*. In case an equation system is non-empty, we omit the trailing* $\emptyset$*.*

Without loss of generality, we use the following naming convention for every equation system of size n:

$$\mathcal{E} = (\sigma_0 X_0(d{:}\, D) = \varphi_0) \ldots (\sigma_{n-1} X_{n-1}(d{:}\, D) = \varphi_{n-1})$$

The syntax above is in *positive normal form*, which means that predicate variables do not occur in the context of an odd number of negations (for simplicity, we only allow negation in the Boolean data terms b). This normal form guarantees that the solution of a PBES (see below) is well-defined.

The set of *bound variables* in an equation system $\mathcal{E}$, denoted $\mathsf{bnd}(\mathcal{E})$, is the set of variable names that occur on the left-hand sides of the equations in $\mathcal{E}$.

We define the *signature* $\mathsf{sig}(X)$ of a predicate variable X to be the product $\{X\} \times \mathbb{D}$. We lift this notion to sets of variables $P \subseteq \mathcal{P}$ and to equation systems in the natural way, *i.e.*, $\mathsf{sig}(P) = \bigcup_{X \in P} \mathsf{sig}(X)$ and $\mathsf{sig}(\mathcal{E}) = \mathsf{sig}(\mathsf{bnd}(\mathcal{E}))$. Elements of a signature type are called *instantiations* in this paper. We denote the instantiation $\langle X, v \rangle$ by $X(v)$, following the equation system syntax.

The *rank* of a predicate variable X_i in an equation system $\mathcal{E}$, denoted $\mathsf{rank}_{\mathcal{E}}(X_i)$ is defined as $\mathsf{rank}_{\mathcal{E}}(X_i) = |\{0 \leq j \leq i \mid \sigma_j \neq \sigma_{j-1}\}|$ if we define $\sigma_{-1} = \nu$.

Definition 2 (Semantics of predicate formulas). *The* semantics *of a predicate formula* φ *is defined in the context of a data environment* δ *and a predicate environment* $\theta: \mathcal{P} \to (\mathbb{D} \to \mathbb{B})$*, which assigns a Boolean valued function to each predicate variable.*

$$[\![b]\!]\theta\delta = [\![b]\!]\delta \qquad [\![X(e)]\!]\theta\delta = \theta(X)([\![e]\!]\delta)$$
$$[\![\varphi \wedge \psi]\!]\theta\delta = [\![\varphi]\!]\theta\delta \wedge [\![\psi]\!]\theta\delta \qquad [\![\varphi \vee \psi]\!]\theta\delta = [\![\varphi]\!]\theta\delta \vee [\![\psi]\!]\theta\delta$$
$$[\![\forall_{d:\, D}\, \varphi]\!]\theta\delta = \forall_{v \in \mathbb{D}}\, [\![\varphi]\!]\theta\delta[v/d] \qquad [\![\exists_{d:\, D}\, \varphi]\!]\theta\delta = \exists_{v \in \mathbb{D}}\, [\![\varphi]\!]\theta\delta[v/d]$$

Although predicate environments are defined as mappings from $\mathcal{P}$ to $\mathbb{D} \to \mathbb{B}$, it is often convenient to think of them as sets of instantiations, *i.e.*, subsets of $\mathcal{P} \times \mathbb{D}$. We will often want to add or remove elements from these sets, which is accomplished using environment updates. For mappings $f\colon \mathbb{D} \to \mathbb{B}$ we define $f[r/v](v) = r$ and $f[r/v](d) = f(d)$ if $d \neq v$. Using this notation, we define an environment update $\theta[r/X(v)]$ as the environment θ in which the instantiation $X(v)$ is mapped to r as follows: $\theta[r/X(v)](X) = \theta(X)[r/v]$ and $\theta[r/X(v)](Y) = \theta(Y)$ if $X \neq Y$. This notation is lifted to sets of instantiations: $\theta[r/S]$ for some set of instantiations $S = \{s_0, s_1, \ldots\}$ is equal to $\theta[r/s_0][r/s_1]\ldots$ (for some arbitrary ordering of the elements of S).

Note that the right hand side φ_i of variable X_i gives rise to the Boolean function $\lambda v \in \mathbb{D}.[\![\varphi_i]\!]\theta\delta[v/d]$. In the following definition, we use this function to construct a predicate transformer T (in our case, a function of type $(\mathbb{D} \to \mathbb{B}) \to (\mathbb{D} \to \mathbb{B})$). Because φ is in positive normal form, it can be shown that T is monotonic over the complete lattice $\langle \mathbb{B}^{\mathbb{D}}, \sqsubseteq \rangle$ (with $\sqsubseteq$ the point-wise lifting of implication), and therefore has least and greatest fixpoints, denoted by μT and νT, respectively.

Definition 3 (Solution). *The* solution *to an interpreted PBES $\langle \mathcal{E}, \theta, \delta \rangle$ is a predicate environment $[\![\mathcal{E}]\!]\theta\delta$ defined inductively as follows (for $\sigma \in \{\mu, \nu\}$):*

$$[\![\emptyset]\!]\theta\delta = \theta$$
$$[\![(\sigma X(d\colon D) = \varphi)\ \mathcal{E}']\!]\theta\delta = [\![\mathcal{E}']\!]\theta[\sigma T/X]\delta,$$

in which T is the predicate transformer associated with X in $\langle \mathcal{E}, \theta, \delta \rangle$*, defined by*

$$T = \lambda f \in \mathbb{B}^{\mathbb{D}}.\ \lambda v \in \mathbb{D}.[\![\varphi]\!]([\![\mathcal{E}']\!]\theta[f/X]\delta)\delta[v/d]\ .$$

We will call a tuple $\langle \mathcal{E}, \theta, \delta \rangle$ consisting of an equation system and associated data and predicate environments an interpreted PBES.

3 Proof Graphs

Computing the solution to a PBES is a hard problem: even for the subclass of BESs, no solving algorithm is known that can solve any BES in polynomial time. For SAT solving, checking that a given solution to the problem is correct is straightforward. While PBES solving is very similar (we are looking for a satisfying assignment to instantiations), it does not share this property. It can be shown that checking that a predicate environment is the solution to a PBES is *as hard* as computing the solution. This also indicates that the solution to a PBES is not in itself enough to explain to a human *why* the encoded problem has a given outcome.

In this section, we show that there is an alternative characterisation of the solution to a PBES, called a *proof graph*, that can be used as a *certificate*. It is in general easier to automatically verify a proof graph than to solve its PBES (it can be checked in polynomial time for BESs), and it offers humans an intelligible explanation of the reason why the solution to a PBES is the way it is.

Our definition of proof graph is based on the notion of *support sets* on Boolean equation systems defined by Tan [13,14]. In the remainder of the text we adopt the convention to write $v^\bullet$ for the postset $\{v' \in V \mid v \to v'\}$ of a vertex v, if the context

provides a set of vertices V and a binary relation $\rightarrow$ on V. The notation is lifted to sets of vertices in the usual way.

Definition 4 (Proof graph). *A* proof graph *for an interpreted PBES $\langle \mathcal{E}, \theta, \delta \rangle$ is defined as a tuple $\langle V, \rightarrow, r \rangle$, where $V \subseteq \mathsf{sig}(\mathcal{E})$, $\rightarrow \subseteq V \times V$, $r \in \mathbb{B}$ and in which for all $X_i(v) \in V$ we have the following:*

- $[\![\varphi_i]\!]\theta[\neg r/\mathsf{sig}(\mathcal{E})][r/X_i(v)^\bullet]\delta[v/d] = r$
- *If $X_i(v) = Z_0(x_0) \rightarrow Z_1(x_1) \rightarrow \ldots$ for some infinite sequence of Z_i and x_i, then $\mathsf{even}(\min\{\mathsf{rank}_{\mathcal{E}}(Z_i) \mid Z_i \in Z^\infty\}) = r$, where Z^∞ is the set of Z_i occurring infinitely often in the sequence.*

We say that a proof graph $\langle V, \rightarrow, r \rangle$ proves *$X_i(v) = r$ if and only if $X_i(v) \in V$.*

In case of a finite proof graph, the second property in Definition 4 amounts to solving the *even-cycle problem* [10]. For certain classes of equation systems, solving the system can be reduced to the even-cycle problem, so there are cases in which checking the proof graph is not faster than solving the equation system. Contrary to solving a BES, however, the even-cycle problem is known to be polynomial: it is sub-quadratic in the size of the BES [7].

Our main theorem states that these graphs can be used as an alternative characterisation of the solution of an equation systems. That is, if a certain instantiation evaluates to r, then there is a proof graph that shows this, and if there is a proof graph that shows a certain instantiation to evaluate to r, then the solution is indeed r for that instantiation.

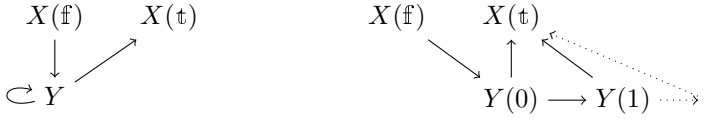
Theorem 1. *Given an interpreted PBES $\langle \mathcal{E}, \theta, \delta \rangle$, we have that for all $X_i(v) \in \mathsf{sig}(\mathcal{E})$, $[\![X_i(v)]\!]\theta\delta = r$ iff there is a proof graph $\langle V, \rightarrow, r \rangle$ on $\langle \mathcal{E}, \theta, \delta \rangle$ such that $X_i(v) \in V$.*

Soundness (the *if*-part of the theorem) and completeness (the *only-if* part) follow from Theorems 2 and 4, presented later in this paper. We first illustrate the concept of a proof graph with an example.

Example 1. Consider the following two equation systems $\mathcal{E}_1$ and $\mathcal{E}_2$, and suppose we are interested in both cases in the value of $X(\mathbb{f})$.

$$\begin{aligned} \mu X(b\colon B) &= Y \vee b & \qquad \mu X(b\colon B) &= Y(0) \vee b \\ \nu Y &= Y \vee (X(\mathbb{t}) \wedge Z) & \qquad \nu Y(n\colon N) &= Y(n+1) \wedge X(\mathbb{t}) \\ \mu Z &= Z \end{aligned}$$

Below are proof graphs for these systems that prove $X(\mathbb{f}) = \mathbb{t}$. Because the equation systems are closed (they do not refer to predicate variable names that are not bound inside the PBES itself), we can assume arbitrary θ and δ. We assume that $[\![\_]\!]\delta$ gives the usual semantics to the function symbols that are used above. Note that $\mathsf{sig}(\mathcal{E}_1) = \{X(d) \mid d \in \mathbb{B}\} \cup \{Y, Z\}$ and $\mathsf{sig}(\mathcal{E}_2) = \{X(d) \mid d \in \mathbb{B}\} \cup \{Y(d) \mid d \in \mathbb{N}\}$.



To see that the graph on the left satisfies the first property of proof graphs for every node, note that the following hold:

$$\begin{aligned}&\text{for } X(\mathbb{f})\text{: } [\![Y \vee b]\!]\theta[\mathbb{f}/\mathsf{sig}(\mathcal{E})][\mathbb{t}/\{Y\}]\delta[\mathbb{f}/b] = \mathbb{t}\\&\text{for } X(\mathbb{t})\text{: } [\![Y \vee b]\!]\theta[\mathbb{f}/\mathsf{sig}(\mathcal{E})]\delta[\mathbb{t}/b] = \mathbb{t}\\&\quad\text{for } Y\text{: } [\![Y \vee X(\mathbb{t})]\!]\theta[\mathbb{f}/\mathsf{sig}(\mathcal{E})][\mathbb{t}/\{Y, X(\mathbb{t})\}]\delta = \mathbb{t}\end{aligned}$$

It also satisfies the second property of proof graphs, for the only predicate variable occurring infinitely often in any infinite sequence of instantiations is Y, and because the rank of Y is 2, indeed $\mathsf{even}(\mathsf{rank}_{\mathcal{E}_1}(Y)) = \mathbb{t}$. Note that in this proof graph, $Y \to X(\mathbb{t})$ can be left out, because the right hand side of the equation for Y is disjunctive and the left disjunct is true. This cannot be done in the right-hand side proof graph, which illustrates how a PBES can give rise to an infinite proof graph. ■

3.1 Soundness

The proof of the main theorem of this section relies heavily on two lemmas that give some insight in how proof graphs work. The first lemma says that the successor relation in a proof graph is a dependency relation: if all successors of an instantiation $X_i(x) \in V$ in a proof graph $\langle V, \to, r\rangle$ can be shown to have value r in the solution of the equation system the graph was defined on, then $X_i(x)$ itself also has value r in the solution.

Lemma 1. *Let $\langle V, \to, r\rangle$ be a proof graph on $\langle \mathcal{E}, \theta, \delta\rangle$, and let $X_i(x) \in V$. Then:*

$$\big(\forall_{X_j(x') \in X_i(x)^\bullet}\ ([\![\mathcal{E}]\!]\theta\delta)(X_j)(x') = r\big) \Rightarrow ([\![\mathcal{E}]\!]\theta\delta)(X_i)(x) = r$$

Let $\mathcal{E}^i$ denote $\mathcal{E}$ without the first i equations. Lemma 2 below shows that, intuitively, one can see predicate environments as assumptions about the unbound variables in a PBES. This gives us an induction principle: the subgraph of a proof graph on $\langle \mathcal{E}, \theta, \delta\rangle$ that contains only instantiations from a subsystem $\mathcal{E}^i$ is a proof graph on $\langle \mathcal{E}^i, \theta', \delta\rangle$, as long as any dependencies on variables outside $\mathcal{E}^i$ are now 'assumed' by θ'.

Lemma 2. *Let $\langle V, \to, r\rangle$ be a proof graph for interpreted equation system $\langle \mathcal{E}, \theta, \delta\rangle$. Let $i < |\mathcal{E}|$, let $V^i = \mathsf{sig}(\mathcal{E}^i) \cap V$, $\to^i = \to \cap (V^i \times V^i)$, and let θ' be any predicate environment such that θ and θ' agree on predicate variables not in $\mathsf{bnd}(\mathcal{E})$, and $\theta'(X_j)(v) = r$ for all $X_j(v) \in {V^i}^\bullet \setminus V^i$. Then $\langle V^i, \to^i, r\rangle$ is a proof graph for $\langle \mathcal{E}^i, \theta', \delta\rangle$.*

Theorem 2. *Let $\langle V, \to, r\rangle$ be a proof graph for interpreted equation system $\langle \mathcal{E}, \theta, \delta\rangle$. Then $([\![\mathcal{E}]\!]\theta\delta)(X_i)(x) = r$ for all $X_i(x) \in V$ ($0 \le i < |\mathcal{E}|$, $x \in \mathbb{D}$).*

Proof (sketch). Let $X_i(x) \in V$; we prove that $([\![\mathcal{E}]\!]\theta\delta)(X_i)(x) = r$ using induction on the size of $\mathcal{E}$ (the outer induction) and on i (the inner induction). The inner induction hypothesis allows us to conclude that $X_j(x) \in V$ implies $([\![\mathcal{E}]\!]\theta\delta)(X_j)(y) = r$ for all $j < i$, and Lemma 2 in combination with the outer induction hypothesis allows us to conclude that $X_j(y) \in V$ implies $([\![\mathcal{E}]\!]\theta\delta)(X_j)(y) = r$ for all $j > i$. (The latter is tricky, because Lemma 2 yields a proof graph on another, derived, PBES, with a different predicate environment.) So, we know that all instantiations $X_j(y)$ with

$j \neq i$ on which the solution of $X_i(x)$ may depend have the value r under $[\![\mathcal{E}]\!]\theta\delta$. If $\sigma_i = \mu$ and $r = \mathbb{t}$, or $\sigma_i = \nu$ and $r = \mathbb{f}$, then assume towards a contradiction that $[\![\mathcal{E}]\!]\theta\delta(X_i)(x) = \neg r$. From the inner induction hypothesis we derive that then $X_i(x) \to X_j(y)$ for some $X_j(y)$ with $j \geq i$ and $[\![\mathcal{E}]\!]\theta\delta(X_j)(y) = \neg r$. We show that we can keep making such steps until we reach $X_j(y)$ with $j = i$ such that again $[\![\mathcal{E}]\!]\theta\delta(X_j)(y) = \neg r$, without visiting $X_j(y)$ with $j < i$. This allows us to construct an infinite path that violates the second condition in Definition 4, which contradicts our assumption that $\langle V, \to, r\rangle$ is a proof graph. If $\sigma_i = \mu$ and $r = \mathbb{f}$ or $\sigma_i = \nu$ and $r = \mathbb{t}$, then we use the fixpoint definition of X_i to show that for all successors $X_j(y)$ of $X_i(x)$ we have $[\![\mathcal{E}]\!]\theta\delta(X_j)(y) = r$, using the well-known result that fixpoints of monotone functions may be approximated by iteratively applying them to some initial approximation. Lemma 1 can then be used to obtain the result. □

3.2 Completeness

Any equation system can be solved by translating it to a parity game and solving that game. We give our proof of completeness by translating equation systems into parity games, and then obtaining proof graphs from these. In practice, parity game solvers prove to be very effective for solving games generated from model checking problems, and tools that perform model checking this way are already available. Our completeness result therefore also yields a practical way of creating proof graphs. We note that there are more methods to solve equation systems, each with their own virtues.

Definition 5 (Parity game). *A parity game is a directed graph $(V, \to, \Omega, \Pi)$, where*

- *V is a set of vertices,*
- *$\to \subseteq V \times V$,*
- *$\Omega\colon V \to F$ is a function that assigns priorities from some finite set $F \subset \mathbb{N}$ to vertices,*
- *$\Pi\colon V \to \{\Diamond, \Box\}$ is a function assigning players to vertices.*

A game starting in a vertex $v \in V$ is played by placing a token on v, and then moving the token along the edges in the graph. Moves are taken indefinitely, or until neither player can move, according to the following simple rule: if the token is on some vertex v, player $\Pi(v)$ moves the token to some vertex w such that $v \to w$. If the result is a finite path, then the winner is the opponent of the last vertex on the path (*i.e.*, the player who cannot do any more moves loses). If the result is an infinite path p in the game graph, then the parity of the lowest priority that occurs infinitely often on p defines the winner of the path. If this priority is even, then player $\Diamond$ wins, otherwise player $\Box$ wins. A *strategy* for player i is a partial function $s\colon V \to V$, that for each vertex owned by player i determines the next vertex to be played onto. By $\Pi(s)$ we denote the player for which the strategy is defined. A strategy s is *winning* from a vertex v iff every path that is the result of playing the game from v according to s, is winning for $\Pi(s)$. Parity games are memoryless determined [5]. From this, it follows that the set of vertices won by player $\Box$ and $\Diamond$ partition V.

When a PBES is translated to a parity game, the nodes of the parity game represent subformulas in the PBES. The logical operators are encoded in the players associated

with the nodes: a node owned by $\Diamond$ represents a disjunction, a node owned by $\Box$ represents a conjunction. The successor relation reflects the subformula relation in the PBES. Predicate variables occurring in right-hand sides of the PBES are simply a reference to a formula (the right-hand side of an equation), and so this formula is treated as a subformula of the equation that contained the reference. This means that predicate variables are not a natural notion in parity games (they are just a syntactic way to represent recursive formulas in a PBES), so this information is lost when translating a PBES to a parity game.

Suppose that we have an equation system $\mathcal{E}$ and a parity game $\langle V, \to, \Omega, \Pi \rangle$, and we have some injective partial function $\pi : V \to \mathsf{sig}(\mathcal{E})$ that gives us this missing information. We can then construct a formula for each node $v \in V$ that expresses its truth value in terms of elements of $\mathsf{sig}(\mathcal{E})$, which can then be evaluated using an environment θ. This evaluation is defined as $[\![v]\!]\theta\pi$:

$$[\![v]\!]\theta\pi = \begin{cases} \forall_{v' \in v^\bullet}\ \mathsf{eval}(v', \theta, \pi), & \Pi(v) = \Box \\ \exists_{v' \in v^\bullet}\ \mathsf{eval}(v', \theta, \pi), & \Pi(v) = \Diamond \end{cases}$$

$$\mathsf{eval}(v, \theta, \pi) = \begin{cases} \theta(\pi(v)), & \text{if } \pi(v) \text{ defined} \\ [\![v]\!]\theta\pi, & \text{otherwise} \end{cases}$$

Definition 6 (Encoding). *We say that a parity game $\langle V, \to, \Omega, \Pi \rangle$* encodes *interpreted equation system $\langle \mathcal{E}, \theta, \delta \rangle$ if and only if there is a partial function $\pi \colon V \to \mathsf{sig}(\mathcal{E})$ such that the following conditions are satisfied:*

1. *For each $\langle v, X_i(x) \rangle \in \pi$ we have*
   - $[\![v]\!]\theta\pi = [\![\varphi_i]\!]\theta\delta[x/d]$,
   - $\Omega(v) = \mathsf{rank}_{\mathcal{E}}(X_i)$.
2. *For all $v, w \in V$, if $v \to w$ and $\pi(w)$ is undefined, then $\Omega(v) \leq \Omega(w)$.*
3. *There is no infinite path in which π is undefined for all nodes on the path.*

Note that the third condition guarantees that $[\![v]\!]\theta\pi$ is well defined: if there would be such infinite paths, then $[\![v]\!]\theta\pi$ may have more than one solution. We choose to resolve this issue by adding this extra restriction to our definition of encoding, keeping our definitions (and proofs) simpler, although it should be possible to drop the third condition and instead modify the definition of $[\![v]\!]\theta\pi$.

Now we have a formal relationship between a PBES and its parity game encoding, we show how a proof graph can be obtained from such an encoding. In essence, we show that there is a direct correspondence between our notion of proof graph, and the notion of a winning strategy in a parity game.

Define $\mathsf{reach}(v, \pi, s)$ as the subset of $\mathsf{sig}(\mathcal{E})$ that is used in the computation of $[\![v]\!]\theta\pi$, restricted to those instantiations that are reachable in the parity game from v by playing according to strategy s.

$$\mathsf{reach}'(v, \pi, s) = \begin{cases} \{\pi(v)\}, & \text{if } \pi(v) \text{ defined} \\ \mathsf{reach}(v, \pi, s) & \text{otherwise} \end{cases}$$

$$\mathsf{reach}(v, \pi, s) = \begin{cases} \bigcup_{v' \in v^\bullet}\ \mathsf{reach}'(v', \pi, s), & \Pi(v) \neq \Pi(s) \\ \mathsf{reach}'(s(v), \pi, s), & \Pi(v) = \Pi(s) \end{cases}$$

Note that again this definition is only properly defined when condition 3 of Definition 6 holds on the parity game.

Suppose $G = \langle V, \to, \Omega, \Pi \rangle$ is an encoding, and s is a strategy in that game that wins from some vertex $v \in V$ for which π is defined. We define a structure $\mathsf{proof}_{G,s}(v) = \langle U_s, \to_s, r \rangle$ where $r \in \mathbb{B}$ and $r = \mathbb{t}$ if and only if $\Pi(s) = \Diamond$, and $U_s \subseteq \mathsf{sig}(\mathcal{E})$ and $\to_s \subseteq U_s \times U_s$ are the smallest sets such that $\pi(v) \in U_s$ and for all $X_i(x) \in U_s$, we have $X_i(x)^\bullet = \mathsf{reach}(\pi^{-1}(X_i(x)), \pi, s)$.

Theorem 3. *Let G be a parity game that is an encoding of $\langle \mathcal{E}, \theta, \delta \rangle$, and let v be a vertex in G such that $\pi(v)$ is defined. If s is a winning strategy from v, then $\mathsf{proof}_{G,s}(v)$ is a proof graph on $\langle \mathcal{E}, \theta, \delta \rangle$.*

We now show that it is always possible to encode a PBES into a parity game. Given is an interpreted PBES $\langle \mathcal{E}, \theta, \delta \rangle$. Let $\mathcal{F}$ be the set of all subformulas of right-hand sides in $\mathcal{E}$. We define a parity game with $V \subseteq (\mathcal{P} \times \mathcal{F} \times (\mathcal{V} \to \mathbb{D}))$ as follows.

The sets V and $\to$ are chosen to be the smallest sets such that $\langle X_i(x), \varphi_i, \delta[x/d] \rangle \in V$ for all $X_i \in \mathsf{bnd}(\mathcal{E})$, and for all $\langle X_i(x), \varphi, \delta' \rangle \in V$ we have the following, based on the shape of φ:

$\varphi = \psi_1 \vee \psi_2$ **or** $\varphi = \psi_1 \wedge \psi_2$, then
- $\langle X_i(x), \psi_1, \delta' \rangle \in V \wedge \langle X_i(x), \varphi, \delta' \rangle \to \langle X_i(x), \psi_1, \delta' \rangle$
- $\langle X_i(x), \psi_2, \delta' \rangle \in V \wedge \langle X_i(x), \varphi, \delta' \rangle \to \langle X_i(x), \psi_2, \delta' \rangle$

$\varphi = \forall_{d' : D}\, \psi$ **or** $\varphi = \exists_{d' : D}\, \psi$, then
- $\forall_{v \in \mathbb{D}}\, \langle X_i(x), \psi, \delta'[v/d'] \rangle \in V \wedge \langle X_i(x), \varphi, \delta' \rangle \to \langle X_i(x), \psi, \delta'[v/d'] \rangle$

$\varphi = X_j(e)$ for some $X_j \in \mathsf{bnd}(\mathcal{E})$ and expression e, then
- $\langle X_j([\![e]\!]\delta'), \varphi_j, \delta'[[\![e]\!]\delta'/d] \rangle \in V \wedge \langle X_i(x), \varphi, \delta' \rangle \to \langle X_j([\![e]\!]\delta'), \varphi_j, \delta'[[\![e]\!]\delta'/d] \rangle$

We define $\Omega(\langle X, \varphi, \delta' \rangle) = \mathsf{rank}_{\mathcal{E}}(X)$. $\Pi(\langle X, \varphi, \delta' \rangle)$ is defined to be $\Diamond$ if φ has the shape $\exists_{d' : D}\, \psi$ or $\psi_1 \vee \psi_2$, and $\Box$ if it has the shape $\forall_{d' : D}\, \psi$ or $\psi_1 \wedge \psi_2$. For $X_j \in \mathsf{bnd}(\mathcal{E})$, we define $\Pi(\langle X, X_j(e), \delta' \rangle) = \Diamond$. For all other nodes, we define $\Pi(\langle X, \varphi, \delta' \rangle) = \Diamond$ if $[\![\varphi]\!]\theta\delta'$ and $\Box$ otherwise.

It is straightforward to check that this translation is an encoding in the sense of Definition 6, as we can define $\pi(\langle X_i(x), \varphi_i, \delta' \rangle) = X_i(x)$.

Properties 2 and 3 follow directly from the fact that in this encoding, if $\langle X_i, \varphi, \delta' \rangle \to \langle X_j(y), \psi, \delta'' \rangle$ and π is not defined for the target, then ψ is a subformula of φ (hence property 3 holds) and $X_i = X_j$ (so property 2 holds). For property 1, note that we created our parity game by 'glueing together' subgraphs that are isomorphic to the φ_i. Using this one-to-one correspondence, the property can be seen to be true by a simple induction on the structure of φ_i.

The following theorem now follows from the fact that we can encode an arbitrary PBES into a parity game, and then recover a proof graph from the parity game using Theorem 3.

Theorem 4. *For every instantiation $X_i(x)$ in some interpreted PBES $\langle \mathcal{E}, \theta, \delta \rangle$, there is a proof graph $\langle V, \to, [\![\mathcal{E}]\!]\theta\delta(X_i)(x) \rangle$ such that $X_i(x) \in V$.*

3.3 Minimality

We have shown that a proof graph contains *enough* information to prove that an instantiation in the graph has a certain solution, but in many cases it would be useful to have a concise proof. We assume that we will be using this definition in the setting where we are given a proof graph G by a PBES solver (not necessarily a parity game solver), and we wish to discard any irrelevant information from the proof. We are therefore looking for a subgraph of this proof graph that is 'as small as possible', in a similar vein to what happens in [2].

Definition 7 (Minimal proof graph). *A proof graph is* minimal *w.r.t. $X_i(v)$ and a proof graph G if and only if it is a subgraph of G, includes $X_i(v)$ and there is no smaller subgraph of G that is a proof graph and includes $X_i(v)$.*

Note that there are two aspects to minimality: no minimal proof graph contains a subgraph that is a proof graph, and minimality with respect to G means that any proof graph contained in G is the same size or larger. Minimising a graph is a difficult problem: we show that the problem is NP-hard by using the same technique as Sahni used in [12] to show that minimising and/or-trees is an NP-complete problem.

Theorem 5. *Given a proof graph G that proves $X_i(v) = r$ (for some r), checking whether a subgraph of size N exists that proves $X_i(v) = r$ is NP-hard.*

Proof. We prove the theorem by reducing CNF-satisfiability to the problem of finding a minimal proof graph. Let a CNF formula φ be given as a set V of variable names, a set C of clause names, and mappings $p : C \to 2^V$ and $n : C \to 2^V$ that give, for each clause, the variables that occur as positive (resp. negative) literals in that clause.

Consider the following PBES:

$$\mu P = \bigwedge_{v \in V} X(v) \wedge \bigwedge_{c \in C} S(c) \qquad \mu S(c\colon C) = \bigvee_{v \in p(c)} T(v) \vee \bigvee_{v \in n(c)} F(v)$$

$$\mu X(v\colon V) = T(v) \vee F(v) \qquad \mu F(v\colon V) = \mathbb{t} \qquad \mu T(v\colon V) = \mathbb{t}$$

Notice that in any proof graph that proves P, we must have the nodes $X(v)$ and $S(c)$ for all $v \in V$ and $c \in C$, as they occur in conjunction in the right-hand side of P. Because of the definition of X, either $T(v)$ or $F(v)$ must be in the graph for every $v \in V$. A proof graph for P therefore contains at least $1 + |C| + 2|V|$ nodes. If it does not contain more nodes, then only $F(v)$ or $T(v)$ is included for all $v \in V$. In that case, the assignment that assigns $\mathbb{f}$ to all nodes for which $F(v)$ occurs in the graph and $\mathbb{t}$ to all nodes for which $T(v)$ occurs in the graph is a satisfying assignment for φ (this can easily be seen from the definition of S). □

Note that if it can be guaranteed that the first requirement in Definition 4 can be checked in polynomial time, then the problem is in fact NP-complete. In the mCRL2 toolkit, the Boolean expressions b in Definition 1 can consist of arbitrary first-order logic formulas, so the checking of proofs is, in the general case, not even decidable.

The problem of minimising finite proof graphs can be reduced to finding minimum prime implicants for Boolean formulas, which was shown to be Σ_2^p-complete in the general setting, but NP-complete for monotone Boolean formulas [6]. This direct connection with proof graphs may mean that algorithms and heuristics for finding minimum prime implicants can be re-used to minimise proof graphs. It is also an interesting connection because in SAT and SMT solving, minimum prime implicants are used as a diagnostic, which is also one of the main applications we have in mind for proof graphs.

Given a proof graph $\langle V, \to, r\rangle$ on $\langle \mathcal{E}, \theta, \delta\rangle$, we will create a formula that encodes the graph in terms of two new sets of variables $\{n_v \mid v \in V\}$ and $\{e_{v\to v'} \mid (v, v') \in \to\}$. Define the translation $\mathsf{tr}_v(V, \delta, \varphi)$ for all $v \in V$ as a function that takes a formula, converts it to a positive normal form formula φ' (pushing negations inwards) and then replaces every $X_j(e)$ or $\neg X_j(e)$ that occurs in φ' by $e_{v\to X_j([\![e]\!]\delta)} \wedge n_{X_j([\![e]\!]\delta)}$ if $X_j([\![e]\!]\delta) \in V$, and by $\mathbb{f}$ otherwise. Using this translation, we define the following Boolean formula for our proof graph.

$$\Phi_{V,\delta} = n_{X_i(v)} \wedge \bigwedge_{X_j(w)\in V} n_{X_j(w)} \Rightarrow \mathsf{tr}_{X_j(w)}(V, \delta[w/d], (r \wedge \varphi_j) \vee (\neg r \wedge \neg\varphi_j)),$$

The formula encodes by the left conjunct the fact that $X_i(v)$ is a node in the proof graph, and the right conjunct encodes the first requirement from Definition 4: if $X_j(w)$ is a node in the proof graph, then its right-hand φ_j side must evaluate to r, if r is substituted for any predicate variable instantiations in φ_j that are also a node in the proof graph. Note that one of the disjuncts that is translated is always ignored, due to r having a fixed value. The $e_{X\to Y}$ variables do not restrict the satisfiability of the formula, as they occur only once. It is easy to see that the formula is satisfiable by filling in $\mathbb{t}$ for all variables.

Theorem 6. *Let I be a minimal prime implicant for $\Phi_{V,\delta}$ and define $V' = \{X_j(w) \in V \mid n_{X_j(w)} \in I\}$ and $\to' = \{(X, Y) \in \to \mid e_{X\to Y} \in I\}$. Then $\langle V', \to', r\rangle$ is a minimal proof graph with respect to $X_i(v)$.*

We now know that we can use techniques from the SAT/SMT community to minimise graphs. It remains an interesting challenge, however, to find a solving algorithm that produces acceptable input for a minimisation algorithm. A proof graph G generated from a parity game strategy, for instance, can easily be seen to be minimal with respect to itself, but will not necessarily be a small graph. A solving method like Gauss elimination for equation systems could yield better input for minimisation, but performs poorly in some cases. It may also be possible to modify parity game solvers like Zielonka's recursive algorithm [15] to generate proof graphs rather than winning strategies.

4 Application

Proof graphs can, like Tan's support sets, be used to certify model checking results: a model checker generates a proof graph, and to ascertain that the result it provided is correct, one can check that this graph is indeed a proof graph. This technique can also be used to 're-use' proofs in model checking: if an equation system $\mathcal{E}$ encodes the model

Algorithm 1. Generation of a PBES for Strong Bisimulation

$$bisim = \nu\{X^{M,S}(d{:}D^M, d'{:}D^S) = match^{M,S}(d,d') \wedge match^{S,M}(d',d),$$
$$X^{S,M}(d'{:}D^S, d{:}D^M) = X^{M,S}(d,d')$$
$$X_a^{p,q}(d{:}D^p, d'{:}D^q, d_a{:}D_a) = step_a^{p,q}(d,d',d_a) \mid a \in Act, (p,q) \in \{(M,S),(S,M)\}\}$$

Where $\nu\mathcal{E}$, for a set of equations $\mathcal{E}$, yields a greatest fixpoint equation system in which $\mathcal{E}$'s equations are ordered according to an arbitrary but total order; we use the following abbreviations, for all $a \in Act, (p,q) \in \{(M,S),(S,M)\}$:

$$match^{p,q}(d{:}D^p, d'{:}D^q) = \bigwedge_{a \in Act} \forall_{e{:}\,E_a^p}(h_a^p(d,e) \Rightarrow X_a^{p,q}(g_a^p(d,e), d', f_a^p(d,e)))$$

$$step_a^{p,q}(d{:}D^p, d'{:}D^q, d_a{:}D_a) = \exists_{e'{:}\,E_a^q} h_a^q(d',e') \wedge (d_a = f_a^q(d',e')) \wedge X^{p,q}(d, g_a^q(d',e'))$$

checking problem of whether some system M satisfies property f, and the behaviour of M is changed only in parts of the state space that do not affect f, then it is usually the case (depending on how straightforward the encoding is) that the proof graph for $\mathcal{E}$ is also a proof graph for the system $\mathcal{E}'$ that reflects the changes in M.

The use of proof graphs is not limited to certification. Tan already indicated that his support sets could be used to obtain counterexamples and witnesses for model checking problems. We show that proof graphs can also be used to derive distinguishing formulas for behavioural equivalence checking problems. For brevity, we focus on strong bisimilarity, but the method we use is not limited to this setting.

We assume that processes are described syntactically using *Linear Process Equations (LPEs)* [8], a representation of (potentially infinite) labelled transition systems. In an LPE, the behaviour is described by a state vector of typed variables, accompanied by a set of condition-action-effect rules. LPEs are typically used in tool sets for process algebraic languages such as mCRL2 and μCRL.

Definition 8. *A* linear process equation *is a parameterised equation of the form*

$$M(d:D) = \sum_{a \in Act} \sum_{e_a : E_a} h_a(d,e_a) \longrightarrow a(f_a(d,e_a)) \cdot M(g_a(d,e_a))$$

where for each parameterised action a taken from a finite set of actions Act, we assume functions f_a, h_a and g_a of type $f_a : D \times E_a \to D_a$, $h_a : D \times E_a \to B$ and $g_a : D \times E_a \to D$. Note that here D, D_a and E_a are general data types and B is the Boolean type. The $\sum$ symbols represent non-deterministic choices.

An LPE M specifies that if in the current state d the condition $h_a(d,e_a)$ holds for any e_a of sort E_a, then an action a carrying data parameter $f_a(d,e_a)$ is possible and the effect of executing this action is the new state $g_a(d,e_a)$. The values of the condition, action parameter and new state may depend on the current state and a non-deterministically chosen value for variable e_a. The operational semantics in terms of labelled transition systems is standard.

Given two LPEs M and S of the form of Definition 8, in which all variables, sorts and functions are indexed with either M or S. The encoding of [1] of whether M is strongly bisimilar to S as an equation system is depicted in Algorithm 1.

The encoding is such that for all θ, δ, $([\![\mathit{bisim}]\!]\theta\delta)(X^{M,S})(v_M, v_S) = \mathbb{t}$ iff $M(v_M)$ is strongly bisimilar to $S(v_S)$. As a consequence, if $M(v_M)$ and $S(v_S)$ are not strongly bisimilar, there is a proof graph $\langle V, \to, \mathbb{f}\rangle$ with $X^{M,S}(v_M, v_S) \in V$ and in which there are no cycles, since *bisim* contains no least fixpoints. Now consider the following transformation, yielding a Hennessy-Milner formula with parameterised actions:

$$\Phi(v) = \begin{cases} \bigwedge\{\Phi(v') \mid v' \in v^\bullet\} & \text{if } v = X^{M,S}(d, d') \\ \neg\Phi(X^{M,S}(d', d)) & \text{if } v = X^{S,M}(d, d') \\ \langle a(d_a)\rangle \bigwedge\{\Phi(v') \mid v' \in v^\bullet\} & \text{if } v = X_a^{M,S}(d, d', d_a) \\ [a(d_a)] \bigvee\{\neg\Phi(v') \mid v' \in v^\bullet\} & \text{if } v = X_a^{S,M}(d, d', d_a) \end{cases}$$

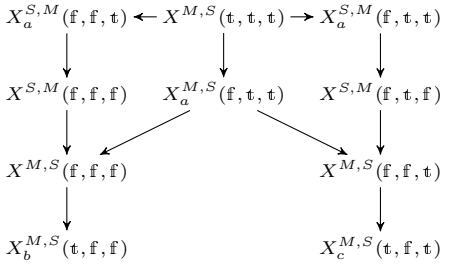
As a convention, the conjunction over an empty set is $\mathbb{t}$ and the disjunction over an empty set is $\mathbb{f}$.

For every pair of non-strongly bisimilar $M(v_M)$ and $S(v_S)$ and all finitely branching proof graphs $\langle V, \to, \mathbb{f}\rangle$ for *bisim* for which $X^{M,S}(v_M, v_S) \in V$, we have $M(v_M) \models \Phi(X^{M,S}(v_M, v_S))$ and $S(v_S) \not\models \Phi(X^{M,S}(v_M, v_S))$. The proof for this is straightforward, using induction on the number of steps to a terminal vertex.

Example 2. Consider two LPEs with actions a, b, c. The LPEs are the prototypical examples of two processes that are not strongly bisimilar due to a difference in the moment of branching.

$$\begin{aligned} &M(v : B) \\ = \;& v \longrightarrow a \cdot M(\mathbb{f}) \\ &+ \neg v \longrightarrow b \cdot M(\mathbb{t}) \\ &+ \neg v \longrightarrow c \cdot M(\mathbb{t}) \end{aligned} \qquad \begin{aligned} &S(t, u : B) \\ = \;& \sum_{u' : B} t \longrightarrow a \cdot S(\mathbb{f}, u') \\ &+ \neg t \wedge u \longrightarrow b \cdot S(\mathbb{t}, \mathbb{t}) \\ &+ \neg t \wedge \neg u \longrightarrow c \cdot S(\mathbb{t}, \mathbb{t}) \end{aligned}$$

Based on the equation system *bisim* for M and S (not given explicitly here), we can construct a proof graph, proving that $M(\mathbb{t})$ and $S(\mathbb{t}, \mathbb{t})$ are not strongly bisimilar. A proof graph for $X^{M,S}(\mathbb{t}, \mathbb{t}, \mathbb{t})$ is depicted below on the left, with a derivation of a distinguishing formula depicted next to it.



$$\begin{aligned} &\Phi(X^{M,S}(\mathbb{t},\mathbb{t},\mathbb{t})) \\ =\;& \Phi(X_a^{M,S}(\mathbb{f},\mathbb{t},\mathbb{t})) \wedge \\ &\Phi(X_a^{S,M}(\mathbb{f},\mathbb{f},\mathbb{t})) \wedge \Phi(X_a^{S,M}(\mathbb{f},\mathbb{t},\mathbb{t})) \\ =\;& \langle a\rangle(\Phi(X^{M,S}(\mathbb{f},\mathbb{f},\mathbb{f})) \wedge \Phi(X^{M,S}(\mathbb{f},\mathbb{f},\mathbb{t}))) \wedge \\ &[a]\Phi(X^{M,S}(\mathbb{f},\mathbb{f},\mathbb{f})) \wedge [a]\Phi(X^{M,S}(\mathbb{f},\mathbb{f},\mathbb{t})) \\ =\;& \langle a\rangle(\Phi(X_b^{M,S}(\mathbb{t},\mathbb{f},\mathbb{f})) \wedge \Phi(X_c^{M,S}(\mathbb{t},\mathbb{f},\mathbb{t}))) \wedge \\ &[a]\Phi(X_b^{M,S}(\mathbb{t},\mathbb{f},\mathbb{f})) \wedge [a]\Phi(X_c^{M,S}(\mathbb{t},\mathbb{f},\mathbb{t})) \\ =\;& \langle a\rangle(\langle b\rangle\mathbb{t} \wedge \langle c\rangle\mathbb{t}) \wedge [a]\langle b\rangle\mathbb{t} \wedge [a]\langle c\rangle\mathbb{t} \end{aligned}$$

Clearly, $M(\mathbb{t}) \models \langle a\rangle(\langle b\rangle\mathbb{t} \wedge \langle c\rangle\mathbb{t}) \wedge [a]\langle b\rangle\mathbb{t} \wedge [a]\langle c\rangle\mathbb{t}$, while $S(\mathbb{t}, \mathbb{t})$ does not.

Note that the proof graph is not minimal. By minimising the proof graph also the distinguishing formula can be minimised. In particular, all vertices of the form $X^{M,S}(\_)$ only need one outgoing edge. ■

5 Related Work

Concepts similar to our proof graphs have been suggested in literature before. For alternation free Boolean equation systems, a similar concept was introduced by Mateescu [11], which he calls *extended Boolean graphs*. The work of Tan and Cleaveland [14], upon which we based our own definition, generalise this to *support sets* on BESs in standard recursive form. Proof graphs in turn generalise support sets.

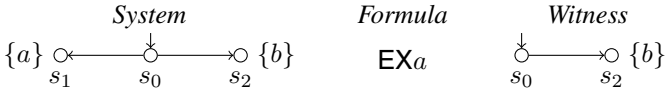
In model checking, it is sometimes possible to generate counterexamples and witnesses that are *linear* (lasso-shaped graphs) or *tree-like* (graphs in which every SCC is a cycle, and of which the component graph is a tree), and that have no subgraph that is still a counterexample or witness [3,2]. A similar thing can be done for proof graphs. We already noted that parity game solvers yield minimal proof graphs. Proof graphs that have the same structure as linear and tree-like counterexamples can be created for purely conjunctive and purely disjunctive PBESs, using the solving method described in [7]. Although it operates on Boolean equation systems rather than PBESs, it can easily be modified to operate directly on PBESs. The algorithm works by solving the *even-cycle problem* [10], which finds reachable cycles with a certain property. It is easy to show that a trace to this cycle plus the cycle itself results in a linear proof graph. This solving method is of particular interest because it allows for efficient solving of LTL formulas via the modal μ-calculus, as was shown in [4]. Because the translation proposed there gives rise to a purely conjunctive PBES, it is possible to obtain linear (and therefore minimal) proof graphs for the LTL model checking problem. For ACTL* problems, the translation can easily be seen to give rise to tree-like proof graphs.

6 Conclusions and Future Work

We have presented an alternative notion of solution to parameterised Boolean equation systems called *proof graphs*. Proof graphs generalise the work in [14] by lifting the syntactic restrictions, and by allowing open equation systems. The use of PBESs rather than BESs gives rise to infinite and infinitely branching proof graphs. This more general setting required us to use a different proof strategy than was used in [13]. Minimisation of proof graphs was shown to be NP-hard (or NP-complete when the data language is decidable in polynomial time), and we showed how proof graphs are related to winning strategies in parity games, and how the problem of minimising proof graphs is related to the problem of finding minimal prime implicants in Boolean formulas. Finally, we have shown how to obtain distinguishing formulas from proof graphs for behavioural equivalence checking problems.

It would be interesting to investigate which solving methods, aside from parity game solvers, can produce proof graphs, and how concise these proof graphs are.

We would also like to relate proof graphs to the more classical notions of ‘counterexample’ and ‘witness’ in model checking. The challenge here is to extract from the proof graph a part of the original model that sufficiently explains the verification result. We like the approach that Tan has taken, by defining sufficient conditions on a proof graph for extracting such counterexamples. Our first investigation in this direction indicates that this is tricky: the conditions that Tan gives allow us to construct an incorrect witness for a system and a CTL formula as shown below.



The problem here is that the proof graph from which the witness is constructed is required to be in some way consistent with the system, but not with the formula. We are currently investigating how this issue can best be resolved, and whether the approach extends to mixed Kripke structures and arbitrary specification languages. This would in particular make it possible to extract witnesses for certain equivalence checking problems (using one of the systems as a specification) in the same way as one would do for model checking.

References

1. Chen, T., Ploeger, B., van de Pol, J., Willemse, T.A.C.: Equivalence checking for infinite systems using parameterized boolean equation systems. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 120–135. Springer, Heidelberg (2007)
2. Clarke, E., Grumberg, O., McMillan, K., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In: DAC 1995, pp. 427–432. ACM (1995)
3. Clarke, E., Jha, S., Lu, Y., Veith, H.: Tree-like counterexamples in model checking. In: LICS, pp. 19–29. IEEE (2002)
4. Cranen, S., Groote, J., Reniers, M.: A linear translation from CTL* to the first-order modal μ-calculus. TCS 412, 3129–3139 (2011)
5. Emerson, E., Jutla, C.: Tree automata, mu-calculus and determinacy. In: FOCS, pp. 368–377. IEEE Computer Society (1991)
6. Goldsmith, J., Hagen, M., Mundhenk, M.: Complexity of DNF minimization and isomorphism testing for monotone formulas. Information and Computation 206(6), 760–775 (2008)
7. Groote, J.F., Keinänen, M.: A sub-quadratic algorithm for conjunctive and disjunctive Boolean equation systems. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 532–545. Springer, Heidelberg (2005)
8. Groote, J., Reniers, M.: Algebraic process verification. In: Bergstra, J., Ponse, A., Smolka, S. (eds.) Handbook of Process Algebra. Elsevier (2001)
9. Groote, J.F., Willemse, T.A.C.: Parameterised boolean equation systems (extended abstract). In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 308–324. Springer, Heidelberg (2004)
10. King, V., Kupferman, O., Vardi, M.Y.: On the complexity of parity word automata. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 276–286. Springer, Heidelberg (2001)
11. Mateescu, R.: Efficient diagnostic generation for Boolean equation systems. In: Graf, S. (ed.) TACAS 2000. LNCS, vol. 1785, pp. 251–265. Springer, Heidelberg (2000)
12. Sahni, S.: Computationally related problems. SICOMP 3(4), 262–279 (1974)
13. Tan, L.: Evidence-Based Verification. PhD thesis, Department of Computer Science, State University of New York (2002)
14. Tan, L., Cleaveland, W.R.: Evidence-based model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 455–470. Springer, Heidelberg (2002)
15. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. TCS 200(1-2), 135–183 (1998)

Generalizing Simulation to Abstract Domains

Vijay D'Silva

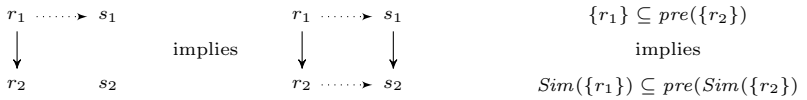
Department of Computer Science,
University of California, Berkeley
vijayd@eecs.berkeley.edu

Abstract. We introduce a notion of subsumption for domains used in abstract interpretation. We show that subsumption has the same properties and applications in the context of abstract interpretation that simulation has for transition systems. These include a modal characterisation theorem, a fixed point characterisation, and the construction of property-preserving abstractions. We use the notion of conjugate functions from algebraic logic to develop bisubsumption, an order-theoretic generalisation of bisimulation to Boolean domains. We prove a representation theorem that relates simulation and subsumption.

1 Spreading the Joy of Simulation

Simulation is a relation between transition systems that has numerous applications in logic, model checking, automata theory, and process algebra. Simulation quotients preserve ACTL and the universal fragment of the μ-calculus [15]. Infinite-state systems with simulation quotients of finite index have decidable model checking problems [11]. Reachability is decidable in systems that are well-founded with respect to a simulation preorder [2,8]. Simulation provides a sufficient, polynomial-time criterion for language inclusion between nondeterministic automata [1,7].

We develop *subsumption*, a lattice-theoretic generalisation of simulation for domains used in abstract interpretation. Subsumption facilitates reasoning about modal logics, property-preservation, and decidability in abstract interpretation in the same manner that simulation facilitates such reasoning over transition systems. We illustrate by recalling the standard definition of simulation below, on the left, and a lattice-theoretic definition below, on the right.



A relation *Sim*, depicted by dotted arrows, is a simulation if for every pair (r_1, s_1) in *Sim* and transition, depicted by solid arrows, from r_1 to r_2, there exists an s_2 such that s_1 transits to s_2 and (r_2, s_2) are in *Sim*. We derive a lattice-theoretic formulation of this definition by observing that the set $\{r_1\}$ is an element of the

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 485–499, 2013.

powerset lattice of states and is contained in the set $pre(\{r_2\})$ of predecessors of r_2. States that simulate r_1 are in the set $Sim(\{r_1\})$ and must have a transition to some state related to r_2, hence must be contained in $pre(Sim(\{r_2\}))$.

We generalise the formulation above to domains in the abstract interpretation sense: complete lattices equipped with monotone functions called *transformers*. Strictly speaking, a domain should have heuristic operators (called widening and narrowing) to enforce convergence of fixed point computations. These are non-monotone and are not considered in logical studies of abstract domains [6,19]. Consider the domains $(A, \sqsubseteq, op^{\mathcal{A}})$ and $(B, \preccurlyeq, op^{\mathcal{B}})$. A *subsumption* is a function $f : A \to B$ satisfying that $a_1 \sqsubseteq op^{\mathcal{A}}(a_2)$ implies $f(a_1) \preccurlyeq op^{\mathcal{B}}(f(a_2))$ for all a_1 and a_2 in A. Subsumption enjoys the standard properties of simulation such as the modal characterisation, fixed point definition, and property preservation indicating that it is an appropriate generalisation of simulation.

Subsumptions generalise simulation in two ways. Instead of a powerset lattice of states, we have a complete lattice that can be non-distributive, and the predecessor operator is replaced by arbitrary monotone functions. Over powerset lattices, using successor operators corresponds to backward simulation, using a composition of successor and predecessor operators yields forward-backward simulations, and nesting a predecessor or successor operator inside a least fixed point, generates stuttering simulation and stuttering backward simulation.

Subsumption is two-steps removed from homomorphisms [18]. A homomorphism h satisfies that 'true implies $h(op^{\mathcal{A}}(a_2)) = op^{\mathcal{B}}(h(a_2))$'. In subsumption, the antecedent true is replaced by $a_1 \sqsubseteq op^{\mathcal{A}}(a_2)$ and the equality is replaced by order. Subsumption shifts focus from preserving the structure of an algebra as in homomorphism to approximating the properties of certain elements.

There are generalisations of simulation to coalgebras [4,12,14]. Transition systems can be represented as Boolean algebras with operators, and we are not aware of a notion of simulation (or bisimulation) in that setting. Coalgebraic characterisation simulation apply to the category of sets or posets [14] but not lattices with operators, as considered here.

Contribution. This paper generalises simulation from transition systems to domains studied in abstract interpretation. Our contributions are:

1. The notions of subsumption and bisubsumption for abstract domains and proofs that the logical, fixed point, and finitary characterisation of simulation and bisimulation carry over to subsumption and bisubsumption. The definition of bisubsumption uses a novel algebraic formulation of bisimulation based on conjugate functions from algebraic logic.
2. Representation theorems relating subsumption and bisubsumption to simulation and bisimulation. Our proofs use representation theorems for Boolean algebras with operators, and distributive lattices with operators.

A copy of the paper with an appendix containing proofs and background material is available at: `http://www.eecs.berkeley.edu/~vijayd/papers/2013/concur.html`

2 Subsumption

We now introduce subsumption and prove a modal characterisation theorem.

Notation. The n-fold composition of a function f with itself is written f^n with f^0 as identity. An *m-termed A-sequence* is a function $\bar{s} : [0, m-1] \to A$. We denote the element $\bar{s}(i)$ as s_i. Function application with $f : A^n \to C$ is written $f(\bar{s})$ with the implicit understanding that $\bar{s}$ is n-termed. In illustrations, we write a set $\{a, b\}$ as a, b to reduce clutter.

Following [6,19], we define domains with respect to a signature. A *signature* is a set of symbols *Sig* with an arity function *ar*. *Constants* are zero arity symbols. A *Sig-domain* $\mathcal{A} = (A, O_A)$ is a complete lattice $(A, \sqsubseteq, \sqcup, \sqcap)$, and a set of monotone functions O_A, called *transformers* such that there is one transformer $op^{\mathcal{A}} : A^{ar(op)} \to A$, for each *op* in *Sig*. A *pointed domain* $(\mathcal{A}, a)$ is an domain with a lattice element a. The element a usually represents initial states or error states. We will drop the word *pointed* for brevity.

Subsumption replaces a transition system by a domain, states by lattice elements, transitions by operators, and simulation by a function between domains. We write $f\langle\bar{a}\rangle$ for the sequence obtained by applying f to each element of $\bar{a}$.

Definition 1. *A function $f : A \to C$ between two domains $\mathcal{A} = (A, O_A)$ and $\mathcal{C} = (C, O_C)$ is a subsumption if the conditions below hold.*

1. *For every constant symbol p, $a \sqsubseteq p^{\mathcal{A}}$ implies that $f(a) \sqsubseteq p^{\mathcal{C}}$.*
2. *For every operator symbol op, and sequence $\bar{a}$ of elements of A of length $ar(op)$, the inequality $a \sqsubseteq op^{\mathcal{A}}(\bar{a})$ implies $f(a) \sqsubseteq op^{\mathcal{C}}(f\langle\bar{a}\rangle)$.*

The domain $(\mathcal{C}, c)$ subsumes $(\mathcal{A}, a)$ if there exists a subsumption $f : A \to C$ such that $c \sqsubseteq f(a)$.

If the domains in question are clear, we say that c subsumes a. We now illustrate how simulation relations generate subsumptions.

Example 1. Two labelled transition systems M and N are shown in Figure 1 along with the domains they generate. The lattices contain powersets of states with constants representing sets of states in which they are true.

$$p^{\mathcal{A}} = \{1\} \qquad q^{\mathcal{A}} = \{2\} \qquad p^{\mathcal{B}} = \{3\} \qquad q^{\mathcal{B}} = \{4, 5\}$$

The domains have predecessor transformers denoted $pre^{\mathcal{A}}$ and $pre^{\mathcal{B}}$. The relation R is a simulation and $f : A \to B$ maps each element of A to its image under R, and has the following properties. Propositions map to propositions: $f(q^{\mathcal{A}}) = f(\{2\}) = \{4, 5\} = q^{\mathcal{B}}$. The simulation condition implies that because $(1, 3)$ is in R and 1 transits to 2, there must be a state s (in this case 4) such that 3 transits to s and $(2, s)$ is in R. When lifted to a domain, we obtain that $\{1\}$ is contained in $pre^{\mathcal{A}}(\{2\})$ and $f(\{1\})$ is not the empty set, so there must be a set of states S, (in this case $f(\{2\})$) such that $f(\{1\}) \subseteq pre^{\mathcal{B}}(S)$ holds. ◁

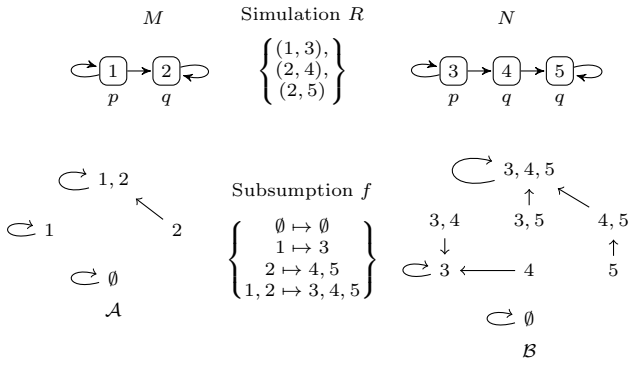


Fig. 1. A simulation and its encoding as a subsumption

Languages. We consider a simple specification language. A *Boolean symbol* is one of $\wedge$, $\vee$, $\neg$, $\bigwedge_i$, $\bigvee_i$, where i is an ordinal indicating the arity of $\bigwedge_i$. We will write $\bigwedge$ and leave the arity implicit. A *language signature* $Sig = Const \cup Mod \cup Bool$, consists of a set *Const* of constants, a set *Mod* of non-null *modalities*, and a set Boolean symbols *Bool*. A *modal signature* has no Boolean symbols.

A *formula* is a constant or the composition $op(\bar{\varphi})$ of an operator with a sequence of formulae of the right arity. *language* L generated by *Sig* contains all formulae over *Sig*. The *conjunctive extension* of L is the language $\mathsf{L}_\wedge$ generated by $Sig \cup \{\wedge\}$. The *completely conjunctive extension* of L is the language $\mathsf{L}_\bigwedge$ generated by $Sig \cup \{\bigwedge\}$, where $\bigwedge$ represents operators of multiple arities.

Formulae over L and (completely) conjunctive extensions of L are interpreted over *Sig*-domains, where *Sig* is a modal signature. Constants are interpreted as lattice elements, modalities as monotone operators, $\wedge$ as $\sqcap$, and $\bigwedge$ as $\bigsqcap$. We define the *interpretation* $[\![\varphi]\!]_\mathcal{A}$ of a formula φ in a domain $\mathcal{A}$, as $p^\mathcal{A}$ for a constant p, and $op^\mathcal{A}([\![\bar{\varphi}]\!]_\mathcal{A})$ for a formula, where $[\![\bar{\varphi}]\!]_\mathcal{A}$ denotes the sequence of elements obtained by interpreting each element of the sequence $\bar{\varphi}$.

Over transition systems, the expression $M, s \models \varphi$ denotes that a state s in M satisfies a formula φ. A pointed domain *satisfies* a formula, denoted $(\mathcal{A}, a) \models \varphi$, if $a \sqsubseteq [\![\varphi]\!]_\mathcal{A}$. We write $\mathsf{L}(\mathcal{A}, a)$ for the set of formulae in L satisfied by $(\mathcal{A}, a)$. A domain $(\mathcal{C}, c)$ *preserves a language* L *with respect to* $(\mathcal{A}, a)$ if $\mathsf{L}(\mathcal{A}, a)$ is contained in $\mathsf{L}(\mathcal{C}, c)$. Two domains are L-*equivalent* if $\mathsf{L}(\mathcal{A}, a) = \mathsf{L}(\mathcal{C}, c)$. Theorem 1 lifts the modal characterisation of simulation to subsumption and domains.

Theorem 1. *Let* $(\mathcal{A}, a)$ *and* $(\mathcal{C}, c)$ *be pointed domains over a modal signature generating a logic* L. *An element* c *subsumes* a *if and only if* $\mathsf{L}_\bigwedge(\mathcal{A}, a) \subseteq \mathsf{L}_\bigwedge(\mathcal{C}, c)$.

Subsumption, being parameterised by a signature, applies to logics over different types of modalities. Our proof is order theoretic and uses only monotonicity and greatest lower bounds unlike standard proofs of simulation, which refer to the

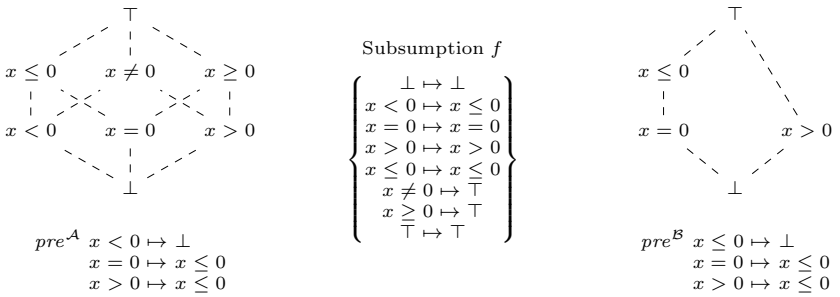


Fig. 2. Two domains that are equivalent with respect to properties specified in a logic closed under conjunction and the next state modality

transition system. The example below shows that subsumption can be used to reason about properties of domains in the same way simulation is used to reason about properties of transition systems.

Example 2. Consider the signature $Sig = \{(x > 0), pre\}$, and the two domains in Figure 2. The constant $(x > 0)$ is interpreted as expected, and *pre* is interpreted as shown, with $pre(x \sqcup y) \mathrel{\hat{=}} pre(x) \sqcup pre(y)$, so values that can be derived this way are omitted. The function f is a subsumption. It also maps elements of A to overapproximations of those elements in B and can be viewed as an abstraction function in the abstract interpretation sense.

Consider a language over $\{(x > 0), \mathsf{EX}_{\rightarrow}, \wedge\}$, with $(x > 0)$ interpreted as expected, $\mathsf{EX}_{\rightarrow}$ interpreted as *pre*, and $\wedge$ as $\sqcap$ in both lattices. The formula $\mathsf{EX}_{\rightarrow}(x > 0)$ is satisfied by the element $(x < 0)$ in $\mathcal{A}$ and by $(x \leq 0)$ in $\mathcal{B}$. Since $f(x < 0)$ is $(x \leq 0)$, and f is a subsumption, we know by Theorem 1 that formulae satisfied by $(\mathcal{A}, (x < 0))$ are satisfied by $(\mathcal{B}, (x \leq 0))$.

Consider a function $g : B \to A$ from elements of B to identically denoted elements in A. This function can be viewed as a concretisation function, forms a Galois connection with f, and is a subsumption. It follows that for every point b in $\mathcal{B}$, there exists a point a in $\mathcal{A}$ such that $(\mathcal{A}, a)$ and $(\mathcal{B}, b)$ satisfy the same formulae in the logic above. While $\mathcal{B}$ is a sound abstraction of $\mathcal{A}$ in the sense of abstract interpretation, we can use subsumption to show that the two satisfy the same logical properties. ◁

Example 3. The domains in Figure 3, over the signature $\{p, q, post\}$, are generated by M and N from Example 1. The arrows on the lattice represent the successor transformer *post*. Consider a signature $\{p, q, \mathsf{EX}_{\leftarrow}\}$, where the previous state modality $\mathsf{EX}_{\leftarrow}$, is interpreted as *post*. The element $\{4\}$ satisfies the formulae $\mathsf{EX}_{\leftarrow}p$, and $\{5\}$ satisfies $\mathsf{EX}_{\leftarrow}q$, but neither element satisfies both formulae. The element $\{2\}$ subsumes $\{4\}$ and $\{5\}$ and satisfies both formulae. ◁

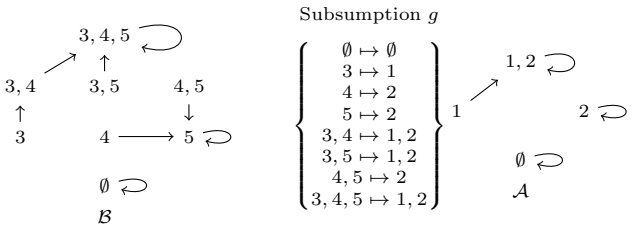


Fig. 3. A subsumption representing a backward simulation

Fixed-Point Characterisation. We characterise subsumption by a fixed point. Given a lattice C, the *pointwise order* $f \sqsubseteq g$ on functions in $A \to C$, holds if $f(x) \sqsubseteq g(x)$ for all x. If C is a complete lattice so is $A \to C$. Our characterisation uses the operators on $A \to C$ defined below. A function f that is not a subsumption is brought closer to one by *ic*, which ensures the condition on constants is satisfied and *iop*, which enforces the condition on transformers.

$$iop(f) \mathrel{\hat{=}} \left\{ a \mapsto \bigsqcap \left\{ op^{\mathcal{C}}(f\langle \bar{a} \rangle) \;\mid\; a \sqsubseteq op^{\mathcal{A}}(\bar{a}) \right\} \right\}$$

$$ic(f) \mathrel{\hat{=}} \left\{ a \mapsto \bigsqcap \left\{ p^{\mathcal{C}} \;\mid\; a \sqsubseteq p^{\mathcal{A}} \right\} \right\} \qquad isub \mathrel{\hat{=}} ic \sqcap iop$$

Theorem 2. *A subsumption is a function satisfying $f \sqsubseteq isub(f)$ and the greatest subsumption is the greatest fixed point* $\mathsf{gfp}(isub)$.

We now apply Theorem 2 to compute a subsumption.

Example 4. Consider the domains in Figure 1. For every f, the function $ic(f)$ is $\{\emptyset \mapsto \emptyset, \{1\} \mapsto \{3\}, \{2\} \mapsto \{4,5\}, \{1,2\} \mapsto \{3,4,5\}\}$. The function $f_0 : A \to B$ that maps all elements of A to $\{3,4,5\}$ is greatest in the pointwise order. Since $pre^{\mathcal{B}}$ maps $\{3,4,5\}$ to itself, the function $iop(f_0)$ maps $\emptyset$ to $\emptyset$ and all other elements of A to $\{3,4,5\}$. The function $f_1 \mathrel{\hat{=}} ic(f_0) \sqcap iop(f_0)$ equals $ic(f_0)$, is a fixed point of $isub(f_1)$, and in fact, is the subsumption in Figure 1. ◁

Stratified Subsumption. Theorem 1 involves infinitary conjunction. Logical equivalence of transition systems in finitary modal languages is characterised by notions weaker than simulation. We weaken subsumption for a similar purpose.

Definition 2. *A sequence of n functions $\bar{f}$ in $A \to C$ is an* n-stratified subsumption *if $f_{i+1} \sqsubseteq f_i$ for all $i < n$ and if the conditions below hold.*

1. *For every constant p and element a in A, $a \sqsubseteq p^{\mathcal{A}}$ implies that $f_0(a) \sqsubseteq p^{\mathcal{C}}$.*
2. *For every operator op, element a and $ar(op)$-sequence $\bar{a}$, if $a \sqsubseteq op^{\mathcal{A}}(\bar{a})$, it holds that $f_n(a) \sqsubseteq op^{\mathcal{C}}(f_{n-1}\langle \bar{a} \rangle)$.*

The pointed domain $(\mathcal{C}, c)$ subsumes $(\mathcal{A}, a)$ up to depth n if there is an n-stratified subsumption $\bar{f}$ such that $c \sqsubseteq f_{n-1}(a)$.

Stratified subsumptions are related to modal languages by modal depth. The *modal depth* of a formula, denoted $mdep(\varphi)$, is the number of nested modal operators in φ, and has the standard definition. Define $\mathsf{L}_n(\mathcal{A}, a)$ to be the set of formulae φ satisfied by a such that $mdep(\varphi) \leq n$. Theorem 3 relates satisfaction of formulae with bounded modal depth and stratified subsumption.

Theorem 3. *Given two pointed domains over a modal signature Sig with finitely many symbols, and the conjunctive extension* L *of the language over Sig,* $\mathsf{L}_n(\mathcal{A}, a) \subseteq \mathsf{L}_n(\mathcal{C}, c)$ *if and only if* $(\mathcal{C}, c)$ *subsumes* $(\mathcal{A}, a)$ *up to depth* n.

For the proof, we introduce a lemma showing that formulae of finite modal depth define finitely many elements of the lattice.

3 Bisubsumption

We now study bisimulation for domains. The different, equivalent definitions of bisimulation suggest different lattice-theoretic formulations. Bisimulation as a simulation whose inverse is a simulation. Subsumptions can replace simulation in this definition, so we only require a notion of inverse. We use conjugate functions from the theory of Boolean Algebras with Operators [13].

Conjugate Functions Two functions $f : A \to C$ and $g : C \to A$ between complete, atomic, Boolean lattices are *conjugate* if, for all elements a in A and c in C, $f(a) \sqcap c = \bot$ exactly if $a \sqcap g(c) = \bot$. Conjugate functions are the lattice-theoretic analogue of the inverse of a relation. The function f is *completely additive* if it satisfies $f(\bigsqcup S) = \bigsqcap f(S)$ for every set $S \subseteq A$. A completely additive function f between complete, atomic Boolean algebras has a unique conjugate [13]. If g is the conjugate, its DeMorgan dual $\neg \circ g \circ \neg$ is the right adjoint of f. For example, the successor operator *post* generated by a transition system is completely additive, has a conjugate *pre* and a right adjoint $\widetilde{pre}$. In logical terms, the past-time modality $\mathsf{EX}_{\leftarrow}$ has a conjugate $\mathsf{EX}_{\rightarrow}$ and right adjoint $\mathsf{AX}_{\rightarrow}$.

A Modal Characterisation We interpret $\neg$ as the complement in a Boolean lattice. If a language contains negation, $a \sqsubseteq [\![\neg\varphi]\!]_{\mathcal{A}}$ exactly if $a \sqsubseteq \neg[\![\varphi]\!]_{\mathcal{A}}$. Preservation requires showing that $\mathsf{L}(\mathcal{A}, a) \subseteq \mathsf{L}(\mathcal{C}, c)$ for properties without a leading negation and $\mathsf{L}(\mathcal{C}, c) \subseteq \mathsf{L}(\mathcal{A}, a)$ for properties with a leading negation.

Definition 3. *A function* $f : A \to C$ *between domains over complete, atomic Boolean lattices is a bisubsumption if* f *is a completely additive subsumption and the conjugate of* f *is a subsumption. Two elements* a *and* c *are in a bisubsumption if there is a bisubsumption* f *with conjugate* b *such that* $c \sqsubseteq f(a)$ *and* $a \sqsubseteq b(c)$.

We emphasise that the definition above is not obvious from the existing definitions of bisimulation for transition systems or coalgebras.

Example 5. Revisit the transition systems and domains in Figure 1. The inverse of R is a simulation, so R is a bisimulation. The conjugate $b : C \to A$ of

f extends the mappings $\{\emptyset \mapsto \emptyset, \{3\} \mapsto \{1\}, \{4\} \mapsto \{2\}, \{5\} \mapsto \{2\}\}$ to satisfy $b(x \cup y) = b(x) \cup b(y)$. See that $f(\{1\}) = \{3\}$, so $f(\{1\}) \cap \{4,5\} = \emptyset$ and conversely, $b(\{4,5\}) = \{2\}$, so $\{1\} \cap b(\{4,5\}) = \emptyset$, as required of conjugate functions. Moreover, b is a subsumption.

Contrast with the domains and language in Example 3, which contain $\mathsf{EX}_{\leftarrow}$. The conjugate of g is $h \mathrel{\hat{=}} \{\emptyset \mapsto \emptyset, \{1\} \mapsto \{3\}, \{2\} \mapsto \{4,5\}, \{1,2\} \mapsto \{3,4,5\}\}$, which is not a subsumption. The inequality $\{2\} \subseteq post^{\mathcal{A}}(\{2\})$ holds but $h(\{2\}) = \{4,5\}$ and is not contained in $post^{\mathcal{A}}(h(\{2\})) = \{5\}$. Over $\{p, q, \mathsf{EX}_{\leftarrow}, \neg, \wedge\}$ formulae, $\{5\}$ satisfies $\neg\mathsf{EX}_{\leftarrow}p$, but $\{2\}$ does not. In classical terms, there is no backward bisimulation (in the sense of [16]) between M and N in Figure 1. $\lhd$

Theorem 4. *Consider a modal signature Sig and the language* L *over* $Sig \cup \{\neg, \bigwedge\}$. *The pointed domains* $(\mathcal{A}, a)$ *and* $(\mathcal{C}, c)$ *are* L*-equivalent if and only if* a *and* c *are in a bisubsumption.*

The proof that a bisubsumption implies equivalence extends that of Theorem 1 with a case for negation, and uses the Dedekind law of conjugate functions [13]. To prove the two elements satisfying the same formulae are in a bisubsumption, we construct two conjugate subsumptions. We introduce a finitary analogue of bisubsumption below and use it to characterise equivalence with respect to formulae of bounded depth.

Definition 4. *Let* $\bar{f}$ *and* $\bar{b}$ *be* $n+1$*-termed sequences of strict additive functions in* $A \to C$ *and* $C \to A$*, respectively such that each* b_i *is the cojugate of the corresponding* f_i. *The sequence* $\bar{f}$ *is a stratified bisubsumption of depth* n *if* $\bar{f}$ *and* $\bar{b}$ *are both stratified subsumptions of depth* n.

Two elements a and c are in an n-stratified bisubsumption if there is an n-stratified bisubsumption $\bar{f}$ with conjugates $\bar{b}$ such that $a \sqsubseteq b_n(c)$ and $c \sqsubseteq f_n(a)$. The inequalities only have to be satisfied by the last functions in the sequence.

Theorem 5. *Given two Sig-algebras over a modal signature with finite symbols, and a language* L *over* $Sig \cup \{\wedge, \neg\}$, $\mathsf{L}_n(\mathcal{A}, a) = \mathsf{L}_n(\mathcal{C}, c)$ *if and only if* c *and* a *are in an* n*-stratified bisubsumption.*

Fixed Point Characterisation. The fixed point characterisations provided earlier extend to bisubsumption. We consider algebras over complete, atomic, Boolean lattices and define operators on $A \to C$ below.

$$
\begin{aligned}
ibc(f) &\mathrel{\hat{=}} \left\{ a \mapsto \bigsqcap \left\{ p^{\mathcal{C}} \;\middle|\; a \sqsubseteq p^{\mathcal{A}} \right\} \cup \left\{ \neg q^{\mathcal{C}} \;\middle|\; a \sqsubseteq \neg q^{\mathcal{A}} \right\} \right\} \\
ibop(f) &\mathrel{\hat{=}} \left\{ a \mapsto \bigsqcap X_a \cup Y_a \right\}, \text{ where} \\
X_a &\mathrel{\hat{=}} \left\{ op^{\mathcal{C}}(f(a')) \;\middle|\; a \sqsubseteq op^{\mathcal{A}}(a') \right\}, \text{ and} \\
Y_a &\mathrel{\hat{=}} \left\{ \neg op^{\mathcal{C}}(f(a')) \;\middle|\; a \sqsubseteq \neg op^{\mathcal{A}}(a') \right\} \\
ibisub &\mathrel{\hat{=}} ic \sqcap iop
\end{aligned}
$$

Theorem 6. *A function* f *is a bisubsumption if and only if* $f \sqsubseteq ibisub(f)$. *The greatest subsumption is the greatest fixed point* $\mathsf{gfp}(ibisub)$.

4 Representation and Abstraction

We now show that subsumption *generates* the notion of simulation. We recall elements of discrete duality theory, which provides a convenient framework for our work.

4.1 Recap of Discrete Duality Theory

The duality theory of modal logic shows that the categories of transition systems, Boolean lattices with transformers, and certain topological spaces, are Stone duals. The term *discrete duality* refers to restrictions under which the topologies are discrete. Discrete duality suffices for this paper.

A Boolean Representation Theorem. Fix a set of constant symbols *Const*. A *labelled transition system* $M = (S, E, prop)$ has a set of states S, a transition relation $E \subseteq S \times S$, and a labelling function $prop : S \to \mathcal{P}(Const)$. A Boolean *pre*-domain $\mathcal{A} = (A, O_A)$ consists of a complete, atomic, Boolean lattice A and a completely additive transformer $pre^{\mathcal{A}}$, which represents a predecessor transformer. The construction below, from [13], maps between transition systems and *pre*-domains. The set $Atom(A)$ contains the atoms of A.

$$\begin{aligned} salg(M) &\mathrel{\hat{=}} \mathcal{M} = (\mathcal{P}(S), O_M) \\ p^{\mathcal{M}} &\mathrel{\hat{=}} \{s \mid p \text{ is in } prop(s)\} \\ pre^{\mathcal{M}} &\mathrel{\hat{=}} \left\{X \mapsto E^{-1}(X)\right\} \end{aligned} \qquad \begin{aligned} srel(\mathcal{A}) &\mathrel{\hat{=}} N = (S_{\mathcal{A}}, E_{\mathcal{A}}, prop_{\mathcal{A}}) \\ S_{\mathcal{A}} &\mathrel{\hat{=}} Atom(A) \\ E_{\mathcal{A}} &\mathrel{\hat{=}} \left\{(a, b) \mid a \sqsubseteq pre^{\mathcal{A}}(b)\right\} \\ prop_{\mathcal{A}} &\mathrel{\hat{=}} \left\{p \mapsto \left\{a \mid a \sqsubseteq p^{\mathcal{A}}\right\}\right\} \end{aligned}$$

Special cases of the theorem below have been repeatedly rediscovered and the earliest account we are aware of is [13].

Theorem 7 ([13]). *For M and $\mathcal{A}$ as above, $srel(salg(M))$ is isomorphic to M, and $\mathcal{A}$ is isomorphic to $salg(srel(\mathcal{A}))$.*

A Distributive Representation Theorem. The definitions that follow are based on [9]. A subset S of a poset P is *join-dense* if every element of P is equal to the join $\bigsqcup Q$ of some subset Q of S. Let L be a bounded lattice. An element $x \neq \bot$ is *completely join-irreducible* if for every subset S of L, $x = \bigsqcup S$ implies that x is in S. As no ambiguity arises, completely join-irreducibles are called join-irreducibles. The set of join-irreducibles of L is $Irr_{\sqcup}(L)$. The set of join-irreducibles below x is $Irr_{\sqcup}(x) \mathrel{\hat{=}} \{y \in Irr_{\sqcup}(L) \mid y \sqsubseteq x\}$.

Powerset lattices and $(\mathbb{N} \cup \{\omega\}, \leq)$, the extension of $\mathbb{N}$ with a greatest upper bound ω, satisfy a strong distributivity property. Consider two sets of indices I and J and the functions $I \to J$. A lattice is *completely distributive* if the identity

$$\bigsqcap \left\{ \bigsqcup \{x_{i,j} \mid j \in J\} \mid i \in I \right\} = \bigsqcup \left\{ \bigsqcap \{x_{i,f(i)} \mid i \in I\} \mid f \in I \to J \right\}$$

is satisfied by every doubly indexed set $\{x_{i,j} \mid i \in I, j \in J\}$. A doubly algebraic distributive lattice (DADL) is a completely distributive lattice in which the completely join-irreducibles are join-dense.

A downset S is a subset of a poset $(M, \preccurlyeq)$ satisfying that if x is in S and $y \preccurlyeq x$ then y is also in S. The set of downsets of M is $\mathcal{D}(M)$. The poset of downsets of M with the subset order, $(\mathcal{D}(M), \subseteq)$, is called the *downset lattice*. Downset lattices are known to be DADLs.

A *monotone transition system* $M = (S, \preccurlyeq, E, prop)$ has a poset of states $(S, \preccurlyeq)$, a transition relation which satisfies $\preccurlyeq \circ E \circ \preccurlyeq \subseteq E$, and a labelling function which satisfies $prop(t) \subseteq prop(s)$ whenever $s \preccurlyeq t$. The order inversion is intentional. A *distributive pre-domain* $\mathcal{A} = (A, O_A)$ consists of a DADL A and a completely additive transformer $pre^{\mathcal{A}}$. The construction below is from [9,3].

$$
\begin{aligned}
salg(M) &\mathrel{\hat{=}} \mathcal{M} = (\mathcal{D}(S), O_M) & srel(\mathcal{A}) &\mathrel{\hat{=}} N = (S_{\mathcal{A}}, E_{\mathcal{A}}, prop_{\mathcal{A}}) \\
p^{\mathcal{M}} &\mathrel{\hat{=}} \{s \mid p \text{ is in } prop(s)\} & S_{\mathcal{A}} &\mathrel{\hat{=}} Irr_{\sqcup}(A) \\
pre^{\mathcal{M}} &\mathrel{\hat{=}} \left\{X \mapsto E^{-1}(X)\right\} & E_{\mathcal{A}} &\mathrel{\hat{=}} \left\{(a,b) \mid a \sqsubseteq pre^{\mathcal{A}}(b)\right\} \\
& & prop_{\mathcal{A}} &\mathrel{\hat{=}} \left\{p \mapsto \left\{a \mid a \sqsubseteq p^{\mathcal{A}}\right\}\right\}
\end{aligned}
$$

Theorem 8 ([9]). *For M and $\mathcal{A}$ as above, $srel(salg(M))$ is isomorphic to M, and $\mathcal{A}$ is isomorphic to $salg(srel(\mathcal{A}))$.*

4.2 Deriving Simulation from Subsumption

We now apply the representation theorems above to generate simulations from subsumptions. We recall the definition of simulation below.

Definition 5. *Let $M_1 = (S_1, E_1, prop_1)$ and $M_2 = (S_2, E_2, prop_2)$ be transition systems with labelling functions mapping states to sets of labels. A relation $Sim \subseteq S_1 \times S_2$ is a labelled simulation if every (r, s) in Sim satisfy the conditions below.*

1. *$prop_1(r) \subseteq prop_2(s)$*
2. *For every state r' of S_1, if (r, r') is in E_1, there exists a state s' of S_2 satisfying that (s, s') is in E_2 and (r', s') is in Sim.*

A relation Sim is a labelled bisimulation if Sim is a simulation and the inverse relation Sim^{-1} is a simulation.

Consider a relation Sim between M_1 and M_2 as above, and a completely additive subsumption f from the Boolean *pre*-domain $\mathcal{A}_1$ to $\mathcal{A}_2$. The relation Sim defines a function between domains and f defines a relation, both shown below.

$$
\begin{aligned}
&salg(Sim) : \mathcal{P}(S_1) \to \mathcal{P}(S_2) && srel(f) \subseteq Atom(A_1) \times Atom(A_2) \\
&salg(Sim) \mathrel{\hat{=}} \{X \mapsto Sim(X)\} && srel(f) \mathrel{\hat{=}} \{(a,b) \mid b \sqsubseteq f(a)\}
\end{aligned}
$$

By the construction above, every simulation can be derived from a subsumption, and every completely additive subsumption between Boolean *pre*-domains can be derived from a simulation. In Definition 1, we defined subsumption as

a monotone function between lattices. The construction above only applies to completely additive subsumptions between complete, atomic, Boolean algebras, so subsumption strictly generalises simulation.

Theorem 9. *Consider Boolean pre-domains $\mathcal{A}_1$ and $\mathcal{A}_2$, and transition systems M_1 and M_2.*

1. *Every completely additive subsumption f from $\mathcal{A}_1$ to $\mathcal{A}_2$ is equivalent to the subsumption $salg(srel(f))$ from $salg(srel(\mathcal{A}_1))$ to $salg(srel(\mathcal{A}_2))$.*
2. *Every simulation Sim from M_1 to M_2 is equivalent to $srel(salg(Sim))$, a simulation from $srel(salg(M_1))$ to $srel(salg(M_2))$.*

The equivalences above hold if simulation is replaced by bisimulation, and subsumption by bisubsumption.

The proof lifts the isomorphism used to prove Theorem 7 to simulations and subsumptions. The proof of Theorem 9 for the bisimulation case requires a result of Jónsson and Tarski, which shows that conjugate functions between Boolean lattices generate a relation and its inverse.

The construction above applies also to labelled transition systems with edge labels. It can be instantiated with different additive operators to obtain different variants of simulation. If we use the operator *post*, we obtain backward simulations [16]. Compositions of operators, such as $post \circ pre$ and $pre \circ post$ are completely additive and yield forward-backward simulations [16]. Recall that pre^n is defined for $n = 0$ as the identity function and for $n = i + 1$ as $pre \circ pre^i$ and that pre^* is the pointwise union of pre^i for all i in $\mathbb{N}$. A subsumption for pre^* represents a stuttering simulation, while one for $post^*$ represents a stuttering backward simulation. The preceding results about subsumption yield modal, fixed point and finitary characterisations of these variants of simulation in the lattice-theoretic setting, and via Theorem 9, in the transition system setting.

Ordered Simulations. By combining representation theorems for distributive lattices with subsumption, we can generate a notion of simulation for monotone transition systems. We have not observed this notion of simulation in the literature despite monotone transition systems being used in infinite-state model checking [2,8] and modal logic [9].

Definition 6. *Given monotone transition systems $M_1 = (S_1, \preccurlyeq_1, E_1, prop_1, act_1)$ and $M_2 = (S_2, \preccurlyeq_2, E_2, prop_2, act_2)$, a relation $Sim \subseteq S_1 \times S_2$ is an ordered simulation if every (r, s) in Sim satisfies the conditions below.*

1. *$prop_1(r) \subseteq prop_2(s)$*
2. *For all states r_1, r_2 of S_1, if $r \preccurlyeq_1 r_1$ and (r_1, r_2) is in E_1 there exist states s_1, s_2 of S_2 satisfying the order $s \preccurlyeq_2 s_1$, with the transition (s_1, s_2) in E_2 and $act_1(r_1, r_2) \subseteq act_2(s_1, s_2)$.*

Ordered simulations represent completely additive subsumptions between distributive *pre*-domains. Consider two monotone *pre*-transition systems M_1 and

M_2 with an ordered simulation Sim from M_1 to M_2. Consider also two distributive *pre*-domains $\mathcal{A}_1$ and $\mathcal{A}_2$ and an additive subsumption f from $\mathcal{A}_1$ to $\mathcal{A}_2$. The constructions below derive a relation from Sim and a function from f.

$$\begin{array}{ll} salg(Sim) : \mathcal{D}(S_1) \to \mathcal{D}(S_2) & srel(f) \subseteq Irr_{\sqcup}(A_1) \times Irr_{\sqcup}(A_2) \\ salg(Sim) \mathrel{\hat{=}} \{X \mapsto Sim(X)\} & srel(f) \mathrel{\hat{=}} \{(a,b) \mid b \sqsubseteq f(a)\} \end{array}$$

The definition of $srel(f)$ is as before with the difference that the relation is defined over join irreducibles. The definition of $salg(Sim)$ is over downwards closed sets instead of sets. The theorem below shows that every additive subsumption between distributive *pre*-domains has a representation as an ordered simulation.

Theorem 10. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be distributive pre-domains and M_1 and M_2 be monotone transition systems.*

1. *Every additive subsumption f from $\mathcal{A}_1$ to $\mathcal{A}_2$ is isomorphic to the subsumption $salg(srel(f))$ from $salg(srel(\mathcal{A}_1))$ to $salg(srel(\mathcal{A}_2))$.*
2. *Every ordered simulation Sim from M_1 to M_2 is isomorphic to $srel(salg(Sim))$, an ordered simulation from $srel(salg(M_1))$ to $srel(salg(M_2))$.*

We emphasise again the generality of subsumption and its utility. It follows from our earlier results about subsumption that ordered simulations have modal and fixed point characterisations and a finitary analogue. In particular, ordered simulation characterises property preservation between monotone transition systems in the logic over the signature $Const \cup \{\mathsf{EX}_{\to}, \bigwedge, \bigvee\}$. Completely additive subsumptions over DADLs can be instantiated to obtain ordered variants of backward, forward-backward, stuttering, and stuttering-backward simulations.

4.3 An Abstract Interpretation Perspective

We conclude the technical treatment with an abstract interpretation perspective on subsumption and bisubsumption. Cousot and Cousot [5] showed that a restricted case of abstract interpretation can be formalised using closure operators. We use closure operators to show that additive subsumptions can be derived as *underapproximations* of subsumptions, and bisubsumptions as underapproximations of additive subsumptions. The operators below are defined on the lattice $A \to C$ of functions with the pointwise order.

$$\begin{aligned} sub(f) &\mathrel{\hat{=}} \bigsqcup \{g \sqsubseteq f \mid g \text{ is a subsumption } \} \\ sadd(f) &\mathrel{\hat{=}} \bigsqcup \{g \sqsubseteq f \mid g \text{ is an additive subsumption } \} \\ bisub(f) &\mathrel{\hat{=}} \bigsqcup \{g \sqsubseteq f \mid g \text{ is a bisubsumption } \} \end{aligned}$$

Theorem 11. *The operators sub, sadd, and bisub are lower closure operators on $A \to C$. For all $f : A \to C$, $sub(f)$ is a subsumption, $sadd(f)$ is a completely additive subsumption, and $bisub(f)$ is a bisubsumption.*

Recall that the image of a complete lattice under a closure operator is a complete lattice. It follows that the set of all subsumptions and bisubsumptions between two domains form a complete lattice. In particular, there are meet and join operations that map a pair of subsumptions to the greatest subsumption that is more precise, and the least subsumption that is less precise, both with respect to the pointwise order.

5 Related Work

It is infeasible to survey the immensely large body of work on simulation in such a short paper. See [20] for a detailed history of bisimulation. The linear-time branching-time papers are the standard references for variants of simulation [21,10], but there exist further variants not covered there [16]. Newer variants are continuously discovered to suit the needs of different applications.

Transition systems are coalgebras, so simulation and bisimulation generalise to coalgebras, as shown in [4,12,14]. Transition systems also generate Boolean lattices with completely additive transformers, and we are not aware of a notion of simulation in this setting. Our work fills this gap providing a notion of simulation and bisimulation for Boolean lattices with operators. Bisimulation is symmetric and characterised by logics with negation, hence bisubsumption is restricted to Boolean lattices. Simulation applies to logics without negation, hence subsumption generalises to lattices with transformers.

The modal languages considered in this paper were closed under all symbols in the signature. The characterisations presented here only apply to languages that strictly adhere to this form. For example, the 2-nested simulation logic of [21], in which formulae have at most one negation, is not of this form. Neither is the common fragment of LTL and CTL identified by [17]. The framework in this paper is not general enough to cover those cases.

Simulation and bisimulation *quotients* have been studied from the perspective of abstract interpretation [19] and coalgebras [14]. In the language of this paper, if $\mathcal{C} = (C, O_C)$ and $\mathcal{A} = (A, O_A)$ are abstract domains with $\rho : C \to C$ an upper closure operator, such that $\rho(C)$ is isomorphic to A, a *forward complete abstraction* $\mathcal{A}$ of $\mathcal{C}$ satisfies $\rho \circ f^{\mathcal{C}} \circ \rho = f^{\mathcal{A}} \circ \rho$. Ranzato and Tapparo [19] showed that simulation and bisimulation *quotients* have constructive characterisations as forward complete abstractions. Their work allows for a lattice-theoretic derivation of simulation and bisimulation quotients. Our work is complementary and provides a lattice-theoretic formulation for comparing two different domains, which need not be related by abstraction functions.

6 Conclusion

Simulation and bisimulation are fundamental notions that have numerous applications in modal logic, model checking and automata theory. In this paper, we introduced subsumption, a generalisation of simulation to domains used in abstract interpretation. We also introduced bisubsumption, a generalisation of bisimulation to a subfamily of domains defined over Boolean lattices. We have shown that the modal, fixed point and finitary characterisations and lattice-structure of the family of simulations and bisimulations all lift to subsumption and bisubsumption.

There are several directions for future work. Simulation-style relations are used to prove impossibility results about properties that cannot be expressed in certain logics. One question is whether subsumption can be used to derive impossibility results about the reasoning capabilities of static analysers. The setting of non-distributive lattices with transformers is used to give semantics to sub-structural logics. A second question is whether subsumption can be used to reason about such structures, particularly to derive abstractions for reasoning about heap manipulating programs with separation logic. The characterisation theorem in this paper applies to modal languages with a simple and restricted grammatical structure. A third question is whether generic techniques can be developed to generate variants of subsumption from the grammar of a logic such that the variant characterises property preservation in the logic. Finally, all the languages considered in this paper are propositional in that they have no first-order structure. The first-order analogue of bisimulation is the Ehrenfeucht-Fraïsse game, for which an algebraic analogue already exists in the form of Fraïsse morphisms. A task left open by this work is to combine subsumption and Fraïsse morphisms to derive first-order subsumptions. Such a notion would enable reasoning about the abstract domains used in static analysers while also providing structural tools for reasoning about sub-structural first-order logics.

Acknowledgements. I thank the anonymous reviewers for their constructive and positive comments despite the apparent simplicity of this work.

References

1. 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: Proc. of Tools and Algorithms for the Construction and Analysis of Systems, pp. 158–174 (2010)
2. Abdulla, P.A., Čerāns, K., Jonsson, B., Tsay, Y.-K.: Algorithmic analysis of programs with well quasi-ordered domains. Information and Computation 160(1-2), 109–127 (2000)
3. Celani, S.A., Jansana, R.: Priestley duality, a Sahlqvist theorem and a Goldblatt-Thomason theorem for positive modal logic. Logic Journal of the IGPL 7(6), 683–715 (1999)
4. Cîrstea, C.: A modular approach to defining and characterising notions of simulation. Information and Computation 204(4), 469–502 (2006)

5. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proc. of Principles of Programming Languages, pp. 269–282. ACM Press (1979)
6. Cousot, P., Cousot, R.: Temporal abstract interpretation. In: Proc. of Principles of Programming Languages, pp. 12–25. ACM Press (2000)
7. Doyen, L., Raskin, J.-F.: Antichain algorithms for finite automata. In: Proc. of Tools and Algorithms for the Construction and Analysis of Systems (2010)
8. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science 256(1-2), 63–92 (2001)
9. Gehrke, M., Nagahashi, H., Venema, Y.: A Sahlqvist theorem for distributive modal logic. Annals of Pure and Applied Logic 131(1-3), 65–102 (2005)
10. van Glabbeek, R.J.: The linear time - branching time spectrum II. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 66–81. Springer, Heidelberg (1993)
11. Henzinger, T.A., Majumdar, R., Raskin, J.-F.: A classification of symbolic transition systems. ACM Transactions on Computational Logic 6(1), 1–32 (2005)
12. Hughes, J., Jacobs, B.: Simulations in coalgebra. Theoretical Computer Science 327(1-2), 71–108 (2004)
13. Jónsson, B., Tarski, A.: Boolean algebras with operators. American Journal of Mathematics 74(1), 127–162 (1952)
14. Levy, P.B.: Similarity quotients as final coalgebras. In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 27–41. Springer, Heidelberg (2011)
15. Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., Bensalem, S.: Property preserving abstractions for the verification of concurrent systems. Formal Methods in Systems Design 6(1), 11–44 (1995)
16. Lynch, N., Vaandrager, F.: Forward and backward simulations I: untimed systems. Information and Computation 121(2), 214–233 (1995)
17. Maidl, M.: The common fragment of CTL and LTL. In: Foundations of Computer Science, p. 643. IEEE Computer Society, Washington, DC (2000)
18. Malacaria, P.: Studying equivalences of transition systems with algebraic tools. Theoretical Computer Science 139(1-2), 187–205 (1995)
19. Ranzato, F., Tapparo, F.: Generalized strong preservation by abstract interpretation. J. of Logic and Computation 17(1), 157–197 (2007)
20. Sangiorgi, D.: On the origins of bisimulation and coinduction. ACM Transactions on Programming Languages and Systems 31(4), 15:1–15:41 (2009)
21. van Glabbeek, R.J.: The linear time – branching time spectrum I. the semantics of concrete, sequential processes. In: Handbook of Process Algebra, pp. 3–99. Elsevier (2001)

Hyperplane Separation Technique for Multidimensional Mean-Payoff Games*

Krishnendu Chatterjee[1] and Yaron Velner[2]

[1] IST Austria (Institute of Science and Technology Austria)
[2] Tel Aviv University, Israel

Abstract. Two-player games on graphs are central in many problems in formal verification and program analysis such as synthesis and verification of open systems. In this work, we consider both finite-state game graphs, and recursive game graphs (or pushdown game graphs) that model the control flow of sequential programs with recursion. The objectives we study are multidimensional mean-payoff objectives, where the goal of player 1 is to ensure that the mean-payoff is non-negative in all dimensions. In pushdown games two types of strategies are relevant: (1) global strategies, that depend on the entire global history; and (2) modular strategies, that have only local memory and thus do not depend on the context of invocation. Our main contributions are as follows: (1) We show that finite-state multidimensional mean-payoff games can be solved in polynomial time if the number of dimensions and the maximal absolute value of the weights are fixed; whereas if the number of dimensions is arbitrary, then the problem is known to be coNP-complete. (2) We show that pushdown graphs with multidimensional mean-payoff objectives can be solved in polynomial time. For both (1) and (2) our algorithms are based on hyperplane separation technique. (3) For pushdown games under global strategies both one and multidimensional mean-payoff objectives problems are known to be undecidable, and we show that under modular strategies the multidimensional problem is also undecidable; under modular strategies the one-dimensional problem is NP-complete. We show that if the number of modules, the number of exits, and the maximal absolute value of the weights are fixed, then pushdown games under modular strategies with one-dimensional mean-payoff objectives can be solved in polynomial time, and if either the number of exits or the number of modules is unbounded, then the problem is NP-hard. (4) Finally we show that a fixed parameter tractable algorithm for finite-state multidimensional mean-payoff games or pushdown games under modular strategies with one-dimensional mean-payoff objectives would imply the fixed parameter tractability of parity games.

1 Introduction

In this work we present a hyperplane separation technique that solves several fundamental algorithmic open questions for multidimensional mean-payoff objectives. We first present an overview of mean-payoff games, then the important extensions, followed by the open problems, and finally our contributions.

* The research was supported by Austrian Science Fund (FWF) Grant No P 23499-N23, FWF NFN Grant No S11407-N23 (RiSE), ERC Start grant (279307: Graph Games), Microsoft faculty fellows award, the RICH Model Toolkit (ICT COST Action IC0901), and was carried out in partial fulfillment of the requirements for the Ph.D. degree of the second author.

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 500–515, 2013.

Mean-Payoff Games on Graphs. Two-player games played on finite-state graphs provide the mathematical framework to analyze several important problems in computer science as well as in mathematics, such as formal analysis of reactive systems [8,24,23]. Games played on graphs are dynamic games that proceed for an infinite number of rounds. The vertex set of the graph is partitioned into player-1 vertices and player-2 vertices. The game starts at an initial vertex, and if the current vertex is a player-1 vertex (resp. player-2 vertex), then player 1 (resp. player 2) chooses an outgoing edge. This process is repeated forever, and gives rise to an outcome of the game, called a *play*, that consists of the infinite sequence of vertices that are visited. The most well-studied payoff criteria in such games is the *mean-payoff* objective, where a weight (representing a reward) is associated with every transition and the goal of one of the players is to maximize the long-run average of the weights; and the goal of the opponent is to minimize. Mean-payoff games and the special case of graphs (with only one player) with mean-payoff objectives have been extensively studied over the last three decades; e.g. [20,14,27,17]. Graphs with mean-payoff objectives can be solved in polynomial time [20], whereas mean-payoff games can be decided in NP $\cap$ coNP [14,27]. The mean-payoff games problem is an intriguing and rare combinatorial problem that lie in NP $\cap$ coNP, but no polynomial time algorithm is known. However, pseudo-polynomial time algorithms exist [27,7]; if the weights are constants the algorithm is polynomial.

The Extensions. Motivated by applications in formal analysis of reactive systems, the study of mean-payoff games has been extended in two directions: (1) pushdown mean-payoff games; and (2) multidimensional mean-payoff games on finite game graphs. Pushdown games, aka games on recursive state machines, can model reactive systems with recursion. Pushdown games have been studied widely with applications in verification, synthesis, and program analysis in [26,1]. In applications of verification and synthesis, the quantitative objectives that typically arise are multidimensional quantitative objectives, e.g., to express properties like the average response time between a grant and a request is below a given threshold ν_1, and the average number of unnecessary grants is below a threshold ν_2. Thus mean-payoff objectives can express properties related to resource requirements, performance, and robustness; multiple objectives can express the different, potentially dependent or conflicting objectives. Moreover, recently many quantitative logics and automata theoretic formalisms have been proposed with mean-payoff objectives in their heart to express properties such as reliability requirements, and resource bounds of reactive systems [9,5,13,4]. Thus pushdown games and graphs with mean-payoff objectives, and finite-state game graphs with multidimensional mean-payoff objectives are fundamental theoretical questions in model checking of quantitative logics and quantitative analysis of reactive systems. Pushdown games with multidimensional objectives are also a natural generalization to study. Furthermore, in applications related to reactive system analysis, the number of dimensions of mean-payoff objectives is typically small, say 2 or 3, as they denote the different types of resources; and the weights denoting the resource consumption amount are also bounded by constants; whereas the state space of the reactive system is huge; see [3,6].

Relevant Aspects of Pushdown Games. In pushdown games two types of strategies are relevant and studied in the literature. The first one are the *global* strategies, where a global strategy can choose the successor vertex depending on the entire global history

of the play; where history is the finite sequence of configurations of the current prefix of a play. The second are *modular* strategies, which are understood more intuitively in the model of games on recursive state machines. A *recursive state machine* (RSM) consists of a set of component machines (or modules). Each module has a set of *nodes* (atomic states) and *boxes* (each of which is mapped to a module), a well-defined interface consisting of *entry* and *exit* nodes, and edges connecting nodes/boxes. An edge entering a box models the invocation of the module associated with the box and an edge leaving the box represents return from the module. In the game version the nodes are partitioned into player-1 nodes and player-2 nodes. Due to recursion the underlying global state-space is infinite and isomorphic to pushdown games. The equivalence of pushdown games and recursive games has been established in [1]. A modular strategy is a strategy that has only local memory, and thus, the strategy does not depend on the context of invocation of the module, but only on the history within the current invocation of the module. Informally, modular strategies are appealing because they are stackless strategies, decomposable into one for each module.

Previous Results and Open Questions. We now summarize the main previous results and open questions and then present our contributions.

1. (Finite-state graphs). Finite-state graphs with mean-payoff objectives can be solved in polynomial time [20], and finite-state graphs with multidimensional mean-payoff objectives can also be solved in polynomial time [25] using the techniques to detect zero-circuits in graphs of [21].

2. (Finite-state games). Finite-state games with a one-dimensional mean-payoff objective can be decided in NP $\cap$ coNP [27,14], and pseudo-polynomial time algorithms exist for mean-payoff games [27,7]: the current fastest known algorithm works in time $O(n \cdot m \cdot W)$, where n is the number of vertices, m is the number of edges, and W is the maximal absolute value of the weights [7]. Finite-state games with multidimensional mean-payoff objectives are coNP-complete with weights in $\{-1, 0, 1\}$ but with arbitrary dimensions [10], and the current best known algorithm works in time $O(2^n \cdot \text{poly}(n, m, \log W))$.

3. (Pushdown graphs and games). Pushdown graphs and games have been studied only for one-dimensional mean-payoff objectives [12]. Under global strategies, pushdown graphs with a one-dimensional mean-payoff objective can be solved in polynomial time, whereas pushdown games are undecidable. Under modular strategies, pushdown graphs with single exit for every module and weights in $\{-1, 0, 1\}$ are NP-hard, and pushdown games with any number of exits and general weight function are in NP [12].

Many fundamental algorithmic questions have remained open for analysis of finite-state and pushdown graphs and games with multidimensional mean-payoff objectives, such as: (A) Can finite-state game graphs with multidimensional mean-payoff objectives with 2 or 3 dimensions and constant weights be solved in polynomial time?; (B) Can pushdown graphs under global strategies with multidimensional mean-payoff objectives be solved in polynomial time?; (C) Can a polynomial time algorithm be obtained for pushdown games under modular strategies with a one-dimensional mean-payoff objective when relevant parameters (such as the number of modules) are bounded?; and (D) In what complexity class does pushdown games under modular strategies with multidimensional mean-payoff objectives lie?

Our Contributions. In this work we present a hyperplane separation technique to provide answers to many of the open fundamental questions. Our contributions are:

1. (Hyperplane technique). We use the separating hyperplane technique from computational geometry to answer the open questions (A) and (B) above. First, we present an algorithm for finite-state games with multidimensional mean-payoff objectives of k-dimensions that works in time $O(n^2 \cdot m \cdot k \cdot W \cdot (k \cdot n \cdot W)^{k^2+2 \cdot k+1})$ (Section 2: Theorem 1), and thus for constant weights and any constant k (not only $k = 2$ or $k = 3$) our algorithm is polynomial. Second, we present a polynomial-time algorithm for pushdown graphs under global strategies with multidimensional mean-payoff objectives (Section 3: Theorem 3); the algorithm is polynomial for general weight function and arbitrary number of dimensions. Our key intuition is to reduce the multidimensional problem to searching for a separating hyperplane such that all realizable mean-payoff vectors lie on one side of the hyperplane. This intuition allows us to search for a vector, which is normal to the hyperplane and reduce the multidimensional problem to one-dimensional problem by multiplying the weight function by the vector.

2. (Modular pushdown games). We first show that the hyperplane techniques do not extend for modular strategies in pushdown games: we show that pushdown games under modular strategies with multidimensional mean-payoff objectives with fixed number of dimensions are undecidable (Section 4: Theorem 4). Thus the only relevant algorithmic problem for pushdown games is the modular strategies problem for a one-dimensional mean-payoff objective; under global strategies even a one-dimensional mean-payoff objective problem is undecidable [12]. It was already shown in [12] that if the number of modules is unbounded, then even with single exits for every module the problem is NP-hard. We show that pushdown games under modular strategies with one-dimensional mean-payoff objectives are NP-hard with two modules and with weights $\{-1, 0, 1\}$ if the number of exits is unbounded (Section 4: Theorem 5). Thus to obtain a polynomial time algorithm we need to bound both the number of modules as well as the number of exits. We show that pushdown games under modular strategies with one-dimensional mean-payoff objectives can be solved in time $(n \cdot \mathsf{M})^{O(\mathsf{M}^5+\mathsf{M}\cdot\mathsf{E}^2)} \cdot W^{O(\mathsf{M}^2+\mathsf{E})}$, where n is the number of vertices, W is the maximal absolute weight, M is the number of modules, and E is the number of exits (Section 4: Theorem 6). Thus if M, E, and W are constants, our algorithm is polynomial. Hence we answer questions (C) and (D).

3. (Hardness for fixed parameter tractability). Given our polynomial time algorithms when the parameters are fixed for finite-state multidimensional mean-payoff games and pushdown games with a one-dimensional mean-payoff objective under modular strategies, a natural question is whether they are fixed parameter tractable, e.g., could we obtain an algorithm that runs in time $f(k) \cdot O(\text{poly}(n, m, W))$ (resp. $f(\mathsf{M}, \mathsf{E}) \cdot O(\text{poly}(n, W))$) for finite-state multidimensional mean-payoff games (resp. for pushdown modular games with one-dimensional objective), for some computable function f (e.g., exponential or double exponential). We show the hardness of fixed parameter tractability problem by reducing the long-standing open problem of fixed parameter tractability of parity games to both the problems (Section 2: Theorem 2 and Section 4: Theorem 7), i.e., fixed parameter tractability of any of the above problems would imply fixed parameter tractability of parity games.

2 Finite-State Multidimensional Mean-Payoff Games

In this section we will present two results: (1) an algorithm for finite-state multidimensional mean-payoff games for which the running time is polynomial when the number of dimensions and weights are fixed; (2) a reduction of finite-state parity games to finite-state multidimensional mean-payoff games with polynomial weights and arbitrary dimensions that shows that fixed parameter tractability of multidimensional mean-payoff games would imply the fixed parameter tractability of parity games.

Game Graphs. A *game graph* $G = ((V, E), (V_1, V_2))$ consists of a *finite* directed graph (V, E) with a finite set V of n vertices and a set E of m edges, and a partition (V_1, V_2) of V into two sets. The vertices in V_1 are *player-1 vertices*, where player 1 chooses the outgoing edges, and the vertices in V_2 are *player-2 vertices*, where player 2 (the adversary to player 1) chooses the outgoing edges. For a vertex $u \in V$, we write $\mathsf{Out}(u) = \{v \in V \mid (u, v) \in E\}$ for the set of successor vertices of u. We assume that every vertex has at least one outgoing edge, i.e., $\mathsf{Out}(u)$ is non-empty for all $u \in V$.

Plays. A game is played by two players: player 1 and player 2, who form an infinite path in the game graph by moving a token along edges. They start by placing the token on an initial vertex, and then they take moves indefinitely in the following way. If the token is on a vertex in V_1, then player 1 moves the token along one of the edges going out of the vertex. If the token is on a vertex in V_2, then player 2 does likewise. The result is an infinite path in the game graph, called *plays*. Formally, a *play* is an infinite sequence $\pi = \langle v_0, v_1, v_2, \ldots \rangle$ of vertices such that $(v_j, v_{j+1}) \in E$ for all $j \geq 0$.

Strategies. A strategy for a player is a rule that specifies how to extend plays. Formally, a *strategy* τ for player 1 is a function $\tau\colon V^* \cdot V_1 \to V$ that, given a finite sequence of vertices (representing the history of the play so far) which ends in a player 1 vertex, chooses the next vertex. The strategy must choose only available successors, i.e., for all $w \in V^*$ and $v \in V_1$ we have $\tau(w \cdot v) \in \mathsf{Out}(v)$. The strategies for player 2 are defined analogously. A strategy is *memoryless* if it is independent of the history and only depends on the current vertex. Formally, a memoryless strategy for player 1 is a function $\tau\colon V_1 \to V$ such that $\tau(v) \in \mathsf{Out}(v)$ for all $v \in V_1$, and analogously for player 2 strategies. Given a starting vertex $v \in V$, a strategy τ for player 1, and a strategy σ for player 2, there is a unique play, denoted $\pi(v, \tau, \sigma) = \langle v_0, v_1, v_2, \ldots \rangle$, which is defined as follows: $v_0 = v$ and for all $j \geq 0$, if $v_j \in V_1$, then $\tau((v_0, v_1, \ldots v_j)) = v_{j+1}$, and if $v_j \in V_2$, then $\sigma((v_0, v_1, \ldots, v_j)) = v_{j+1}$.

Graphs Obtained under Memoryless Strategies. A player-1 graph is a special case of a game graph where all vertices in V_2 have one successor (and player-2 graphs are defined analogously). Given a memoryless strategy σ for player 2, we denote by G^σ the player-1 graph obtained by removing from all player-2 vertices the edges not chosen by σ.

Multidimensional Mean-Payoff Objectives. For multidimensional mean-payoff objectives we will consider game graphs along with a weight function $w : E \to \mathbb{Z}^k$ that maps each edge to a vector of integer weights. We denote by W the maximal absolute value of the weights. For a finite path π, we denote by $w(\pi)$ the sum of the weight vectors of the edges in π and $\mathsf{Avg}(\pi) = \frac{w(\pi)}{|\pi|}$, where $|\pi|$ is the

length of π, denotes the average vector of the weights. We denote by $\mathsf{Avg}_i(\pi)$ the projection of $\mathsf{Avg}(\pi)$ to the i-th dimension. For an infinite path π, let ρ_t denote the finite prefix of length t of π; and we define $\mathsf{LimInfAvg}_i(\pi) = \liminf_{t\to\infty} \mathsf{Avg}_i(\rho_t)$ and analogously $\mathsf{LimSupAvg}_i(\pi)$ with $\liminf$ replaced by $\limsup$. For an infinite path π, we denote by $\mathsf{LimInfAvg}(\pi) = (\mathsf{LimInfAvg}_1(\pi), \ldots, \mathsf{LimInfAvg}_k(\pi))$ (resp. $\mathsf{LimSupAvg}(\pi) = (\mathsf{LimSupAvg}_1(\pi), \ldots, \mathsf{LimSupAvg}_k(\pi))$) the limit-inf (resp. limit-sup) vector of the averages (long-run average or mean-payoff objectives). The objective of player 1 we consider is to ensure that the mean-payoff is non-negative in every dimension, i.e., to ensure $\mathsf{LimInfAvg}(\pi) \geq \mathbf{0}$, where $\mathbf{0}$ denotes the vector of all zeros. A mean-payoff objective is invariant to the shift operation, i.e., if in a dimension i, we require that the mean-payoff is at least ν_i, then we subtract ν_i in the weight vector from every edge in the i-th dimension and require the mean-payoff is at least 0 in dimension i. Hence WLOG the comparison is with $\mathbf{0}$. We will present all the results for $\mathsf{LimInfAvg}$ objectives and the results for $\mathsf{LimSupAvg}$ objectives are simpler. In sequel we will write LimAvg for $\mathsf{LimInfAvg}$. Also all the results we will present would hold if we replace the non-strict inequality ($\geq \mathbf{0}$) with a strict inequality ($> \mathbf{0}$).

Winning Strategies. A player-1 strategy τ is a winning strategy from a set U of vertices, if for all player-2 strategies σ and all $v \in U$ we have $\mathsf{LimAvg}(\pi(v, \tau, \sigma)) \geq \mathbf{0}$. A player-2 strategy is a winning strategy from a set U of vertices if for all player-1 strategies τ and for all $v \in U$ we have that the path $\pi(v, \tau, \sigma)$ does not satisfy $\mathsf{LimAvg}(\pi(v, \tau, \sigma)) \geq \mathbf{0}$. The winning region for a player is the largest set U such that the player has a winning strategy from U.

Intuition and Key Ideas. Our key insight to solve multidimensional mean-payoff games is to search for a hyperplane $\mathcal{H}$ such that player 2 can ensure a mean-payoff vector below $\mathcal{H}$. Intuitively, we show that if such a hyperplane exists, then any vector below $\mathcal{H}$ is negative in at least one dimension, and thus the multidimensional mean-payoff objective for player 1 is violated. Conversely, we show that if for all hyperplanes $\mathcal{H}$ player 1 can achieve a mean-payoff vector that lies above $\mathcal{H}$, then player 1 can ensure the multidimensional mean-payoff objective. The technical argument relies on the fact that if we have an infinite sequence of unit vectors $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots$ and $\boldsymbol{b}_\ell$ lies above the hyperplane that is normal to $\sum_{j=1}^{\ell-1} \boldsymbol{b}_j$, then $\liminf_{\ell\to\infty} \frac{1}{\ell} \cdot \sum_{j=1}^{\ell} \boldsymbol{b}_j = \mathbf{0}$.

Multiple Dimensions to One Dimension. Given a multidimensional weight function w and a vector $\boldsymbol{\lambda}$, we denote by $w \cdot \boldsymbol{\lambda}$ the one-dimensional weight function that assigns every edge e the weight value $w(e)^T \cdot \boldsymbol{\lambda}$, where $w(e)^T$ is the transpose of the weight vector $w(e)$. We show that with the hyperplane technique we can reduce a game with multidimensional mean-payoff objective to the same game with a one-dimensional mean-payoff objective. A vector $\boldsymbol{b}$ lies above a hyperplane $\mathcal{H}$ if $\boldsymbol{\lambda}$ is the normal vector of $\mathcal{H}$ and $\boldsymbol{b}^T \cdot \boldsymbol{\lambda} \geq 0$. Hence, player 1 can achieve a mean-payoff vector that lies above $\mathcal{H}$ if and only if player 1 can ensure the one-dimensional mean-payoff objective with weight function $w(e) \cdot \boldsymbol{\lambda}$.

Examples. Consider the game graph G_1 (Figure 1) where all vertices belong to player 1. The weight function w_1 labels each edge with a two-dimensional weight vector. In G_1, player 1 can ensure all mean-payoff vectors that are convex combination of $(1, -2), (-2, 1)$ and $(-1, -1)$ (see Figure 3). All the vectors reside below the hyperplane $y = -x$, and consider the normal vector $\boldsymbol{\lambda} = (1, 1)$ to the hyperplane $y = -x$.

All the cycles in G_1 with weight function $w_1 \cdot \boldsymbol{\lambda}$ have negative weights. Therefore player 1 loses in the one-dimensional mean-payoff objective. Consider the game graph G_2 (Figure 2) with all player-1 vertices; where player 1 can achieve any mean-payoff vector that is a convex combination of $(2,-1),(-1,2)$ and $(-2,-1)$ (see Figure 4). Every two-dimensional hyperplane that passes through the origin intersects with the feasible region. Thus, no separating hyperplane exists.

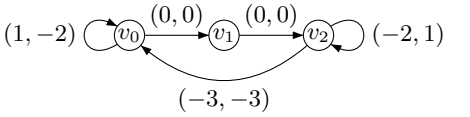


Fig. 1. Game graph G_1

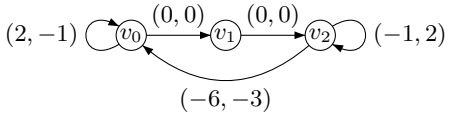


Fig. 2. Game graph G_2

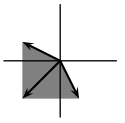


Fig. 3. Feasible vectors for G_1

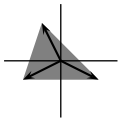


Fig. 4. Feasible vectors for G_2

Basic Lemmas and Assumptions. We now prove two lemmas to formalize the intuition related to reduction to one-dimensional mean-payoff games. Lemma 1 requires two assumptions, which we later show (in Lemma 4) how to deal with. The assumptions are as follows: (1) The first assumption (we refer Assumption 1) is that every outgoing edge of player-2 vertices is to a player-1 vertex; formally, $E \cap (V_2 \times V) \subseteq E \cap (V_2 \times V_1)$. (2) The second assumption (we refer Assumption 2) is that every player-1 vertex has k self-loop edges $e_1, \ldots, e_k$ such that $w_i(e_j) = 0$ if $i \neq j$ and $w_i(e_i) = -1$. Let us denote by Win^2 the player-2 winning region in the multidimensional mean-payoff game with weight function w, and by $\mathsf{Win}^2_{\boldsymbol{\lambda}}$ the player-2 winning region in the one-dimensional mean-payoff game with the weight function $w \cdot \boldsymbol{\lambda}$. Lemma 1 shows that if $\mathsf{Win}^2_{\boldsymbol{\lambda}} \neq \emptyset$, then $\mathsf{Win}^2 \neq \emptyset$; thus gives a sufficient condition for non-emptiness of Win^2. Lemma 2 complements Lemma 1, and does not require Assumption 1 or Assumption 2.

Lemma 1. *Given a game graph G that satisfies Assumption 1 and Assumption 2, and a multidimensional mean-payoff objective with weight function w, for every $\boldsymbol{\lambda} \in \mathbb{R}^k$ we have $\mathsf{Win}^2_{\boldsymbol{\lambda}} \subseteq \mathsf{Win}^2$; (hence, if $\mathsf{Win}^2_{\boldsymbol{\lambda}} \neq \emptyset$, then $\mathsf{Win}^2 \neq \emptyset$).*

Proof. Let σ be a player-2 winning strategy in G from an initial vertex v_0 for the mean-payoff objective with weight function $w \cdot \boldsymbol{\lambda}$. Due to Assumption 1 and Assumption 2 it follows that $\boldsymbol{\lambda} \in (0,\infty)^k$. We show σ is also a player-2 winning strategy wrt the multidimensional mean-payoff objective. Let ρ be a play that is consistent with σ. Since σ is a player-2 winning strategy for the mean-payoff objective with weight function $w \cdot \boldsymbol{\lambda}$, there exists a constant $c > 0$ such that there are infinitely many prefixes of ρ with average weight (according to $w \cdot \boldsymbol{\lambda}$) at most $-c$. Let $\lambda_{\min} = \min\{\boldsymbol{\lambda}_i \mid 1 \leq i \leq k\}$ be the minimum value of $\boldsymbol{\lambda}$ among its dimension. Since $\boldsymbol{\lambda} \in (0,\infty)^k$, it follows that $\lambda_{\min} > 0$. There must be a dimension i with infinitely many prefixes of ρ with average weight at most $-\frac{c \cdot \lambda_{\min}}{k} < 0$. Hence the multidimensional objective is violated. □

Lemma 2. *Given a game graph G and a multidimensional mean-payoff objective with weight function w, if for all $\boldsymbol{\lambda} \in \mathbb{R}^k$ we have $\mathsf{Win}_{\boldsymbol{\lambda}}^2 = \emptyset$, then we have $\mathsf{Win}^2 = \emptyset$.*

Proof. Since $\mathsf{Win}_{\boldsymbol{\lambda}}^2 = \emptyset$ for every $\boldsymbol{\lambda} \in \mathbb{R}^k$, it follows by the determinacy of one-dimensional mean-payoff games [14] that for all $\boldsymbol{\lambda} \in \mathbb{R}^k$, player 1 can ensure the one-dimensional mean-payoff objective with weight function $w \cdot \boldsymbol{\lambda}$ by a memoryless strategy $\tau_{\boldsymbol{\lambda}}$ in G (from all initial vertices). We now present an explicit construction of a player-1 winning strategy for the multidimensional mean-payoff objective in G. We construct a player-1 winning strategy τ for the multidimensional objective in the following way:

- Initially, set $\boldsymbol{b}_0 := (1, 1, \ldots, 1)$.
- For $i = 1, 2, \ldots, \infty$, in iteration i play as follows:
  - Set $\boldsymbol{\lambda}_{b_i} := -\boldsymbol{b}_{i-1}$. In τ, player 1 plays according to $\tau_{\boldsymbol{\lambda}_{b_i}}$ for i rounds.
  - Let ρ_i be the play suffix that was formed in the last i rounds (or steps) of the play. From ρ_i we obtain the part of ρ_i that consists of cycles (that are possibly repeated) and denote the part as ρ_i^2; and an acyclic part ρ_i^1 of length at most n.
  - Set $\boldsymbol{b}_i := \boldsymbol{b}_{i-1} + w(\rho_i^2)$; and proceed to the next iteration.

In order to prove that τ is a winning strategy, it is enough to prove that for every play ρ that is consistent with τ, the Euclidean norm of the average weight vector tends to zero as the length of the play tends to infinity.

We first compute the Euclidean norm of $\boldsymbol{b}_i$. For this purpose we observe that $\tau_{\boldsymbol{\lambda}_{b_i}}$ is a memoryless winning strategy for the one-dimensional mean-payoff game with weight function $w \cdot \boldsymbol{\lambda}_{b_i}$; and hence it follows that for every cycle C in the graph $G^{\tau_{\boldsymbol{\lambda}_{b_i}}}$ the sum of the weights of C according to $w \cdot \boldsymbol{\lambda}_{b_i}$ is non-negative. Since ρ_i^2 is composed of cyclic paths, we must have $w(\rho_i^2)^T \cdot \boldsymbol{\lambda}_{b_i} \geq 0$; and hence, we have $w(\rho_i^2)^T \cdot \boldsymbol{b}_{i-1} \leq 0$. Thus

$$|\boldsymbol{b}_i| = |\boldsymbol{b}_{i-1} + w(\rho_i^2)| = \sqrt{|\boldsymbol{b}_{i-1}|^2 + 2 \cdot w(\rho_i^2)^T \cdot \boldsymbol{b}_{i-1} + |w(\rho_i^2)|^2} \leq \sqrt{|\boldsymbol{b}_{i-1}|^2 + |w(\rho_i^2)|^2}$$

Since W is the maximal absolute value of the weights, it follows that $W \cdot \sqrt{k}$ is a bound on the Euclidean norm of any average weight vector. Since the length of ρ_i^2 is at most i we get that $|\boldsymbol{b}_i| \leq \sqrt{|\boldsymbol{b}_{i-1}|^2 + k \cdot W^2 \cdot i^2} \leq \sqrt{k \cdot W^2 \cdot i^3}$. We are now ready to compute the the Euclidean norm of the play after the i-th iteration. We denote the weight vector after the i-th iteration by $\boldsymbol{x}_i$ and observe that $\boldsymbol{x}_i = \boldsymbol{b}_i + \sum_{j=1}^{i} w(\rho_j^1)$ and by the Triangle inequality we get that $|\boldsymbol{x}_i| \leq |\boldsymbol{b}_i| + \sum_{j=1}^{i} |w(\rho_j^1)|$. Since the length of ρ_i^1 is at most n and by the bound we obtained over $\boldsymbol{b}_i$ we get that $|\boldsymbol{x}_i| \leq \sqrt{k \cdot W^2 \cdot i^3} + i \cdot n \cdot W \cdot \sqrt{k}$. For a position j of the play between iteration i and iteration $i+1$, let us denote by $\boldsymbol{y}_j$ the weight vector after the play prefix at position j. Since there are i steps played in iteration i we have $|\boldsymbol{y}_j| \leq |\boldsymbol{x}_i| + i \cdot W \cdot \sqrt{k}$. Finally, since after the $(i-1)$-th iteration $\sum_{t=1}^{i-1} t = i \cdot (i-1)/2$ rounds were played, we get that the Euclidean norm of the average weight vector, namely, $|\frac{\boldsymbol{y}_j}{j}| \leq \frac{|\boldsymbol{y}_j|}{i \cdot (i-1)/2}$, tends to zero as i tends to infinity. It follows that the limit average of the weight vectors is zero. □

Lemma 1 and Lemma 2 suggest that in order to check if player-2 winning region is non-empty in a multidimensional mean-payoff game it suffices to go over all (uncountably many) $\boldsymbol{\lambda} \in \mathbb{R}^k$ and check whether player-2 winning region is non-empty in the

one-dimensional mean-payoff game with weight function $w \cdot \boldsymbol{\lambda}$. Lemma 3 shows that we need to consider only finitely many vectors.

Notations. For the rest of this section, we denote $M = (k \cdot n \cdot W)^{k+1}$, where W is the maximal absolute value of the weight function. For a positive integer ℓ, we will denote by $\mathbb{Z}_\ell^{\pm} = \{i \mid -\ell \leq i \leq \ell\}$ (resp. $\mathbb{Z}_\ell^{+} = \{i \mid 1 \leq i \leq \ell\}$) the set of integers (resp. positive integers) from $-\ell$ to ℓ.

Lemma 3. *Let G be a game graph with a multidimensional mean-payoff objective and a weight function w. There exists $\boldsymbol{\lambda}_0 \in \mathbb{R}^k$ for which player-2 winning region is non-empty in G for the one-dimensional mean-payoff objective with weight function $w \cdot \boldsymbol{\lambda}_0$ if and only if there exists $\boldsymbol{\lambda} \in (\mathbb{Z}_M^{\pm})^k$ such that the player-2 winning region is non-empty in G for the one-dimensional mean-payoff objective with weight function $w \cdot \boldsymbol{\lambda}$.*

Proof. Suppose that player 2 has a memoryless winning strategy σ in G from an initial vertex v_0 for the one-dimensional mean-payoff objective with weight function $w \cdot \boldsymbol{\lambda}_0$. Let $C_1, \ldots, C_m$ be the simple cycles that are reachable from v_0 in the graph G^σ. Since σ is a player-2 winning strategy it follows that $w(C_i)^T \cdot \boldsymbol{\lambda}_0 < 0$ for every $i \in \{1, \ldots, m\}$. We note that for all $1 \leq i \leq m$ we have $w(C_i) \in (\mathbb{Z}_{n \cdot W}^{\pm})^k$ (since C_i is a simple cycle, in every dimension the sum of the weights is between $-n \cdot W$ and $n \cdot W$). Then by [22, Lemma 2, items c and d] it follows that there is a vector of integers $\boldsymbol{\lambda}$ such that $w(C_i)^T \cdot \boldsymbol{\lambda} \leq -1 < 0$, for all $1 \leq i \leq m$; and $\boldsymbol{\lambda} \in (\mathbb{Z}_M^{\pm})^k$. Since all the reachable cycles from v_0 in G^σ are negative according to $w \cdot \boldsymbol{\lambda}$, we get that σ is a winning strategy for the one-dimensional mean-payoff game with weight function $w \cdot \boldsymbol{\lambda}$; and hence the proof for the direction from left to right follows. The converse direction is trivial. □

Lemma 4 removes the two assumptions of Lemma 1; the technical proof (in [11]) requires Lemma 1, Lemma 2, and Lemma 3.

Lemma 4. *Let G be a game graph with a multidimensional mean-payoff objective with a weight function w. The following assertions hold: (1) $\bigcup_{\boldsymbol{\lambda} \in (\mathbb{Z}_M^{+})^k} \mathsf{Win}_{\boldsymbol{\lambda}}^2 \subseteq \mathsf{Win}^2$. (2) If $\bigcup_{\boldsymbol{\lambda} \in (\mathbb{Z}_M^{+})^k} \mathsf{Win}_{\boldsymbol{\lambda}}^2 = \emptyset$, then $\mathsf{Win}^2 = \emptyset$.*

Attractor Removal. To use the result of Lemma 4 iteratively to solve finite-state games with multidimensional mean-payoff objectives, we need the notion of *attractors*. For a set U of vertices, $Attr_2(U)$ is defined inductively as follows: $U_0 = U$ and for all $i \geq 0$ we have $U_{i+1} = U_i \cup \{v \in V_1 \mid \mathsf{Out}(v) \subseteq U_i\} \cup \{v \in V_2 \mid \mathsf{Out}(v) \cap U_i \neq \emptyset\}$, and $Attr_2(U) = \bigcup_{i \geq 0} U_i$. Intuitively, from U_{i+1} player 2 can ensure to reach U_i in one step against all strategies of player 1, and thus $Attr_2(U)$ is the set of vertices such that player 2 can ensure to reach U against all strategies of player 1 in finitely many steps. The set $Attr_2(U)$ can be computed in linear time [18,2]. Observe that if G is a game graph, then for all U, the game graph induced by the set $V \setminus Attr_2(U)$ is also a game graph (i.e., all vertices in $V \setminus Attr_2(U)$ have outgoing edges in $V \setminus Attr_2(U)$). In multidimensional mean-payoff games, if U is a set of vertices such that player 2 has a winning strategy from every vertex in U, then player 2 has a winning strategy from all vertices in $Attr_2(U)$, and we can recurse in the game graph after removal of $Attr_2(U)$.

Algorithm. We now present our iterative algorithm that is based on Lemma 4 and attractor removal. In the current iteration i of the game graph execute the following steps:

sequentially iterate over vectors $\boldsymbol{\lambda} \in (\mathbb{Z}_M^+)^k$; and if for some $\boldsymbol{\lambda}$ we obtain a non-empty set U of winning vertices for player 2 for the one-dimensional mean-payoff objective with weight function $w \cdot \boldsymbol{\lambda}$ in the current game graph, remove $Attr_2(U)$ from the current game graph and proceed to iteration $i+1$. Otherwise if for all $\boldsymbol{\lambda} \in (\mathbb{Z}_M^+)^k$, player 1 wins from all vertices for the one-dimensional mean-payoff objective with weight function $w \cdot \boldsymbol{\lambda}$, then the set of current vertices is the set of winning vertices for player 1. The correctness of the algorithm follows from Lemma 4 and attractor removal. Since one-dimensional mean-payoff games with n vertices, m edges, and maximal weight W can be solved in time $O(n \cdot m \cdot W)$ [7], we obtain the following result.

Theorem 1. *The set of winning vertices for player 1 in a multidimensional mean-payoff game with n vertices, m edges, k-dimensions, and maximal absolute weight W can be computed in time $O(n^2 \cdot m \cdot k \cdot W \cdot (k \cdot n \cdot W)^{k^2+2 \cdot k+1})$.*

Hardness for Fixed Parameter Tractability. We reduce finite-state parity games to finite-state multidimensional mean-payoff games with weights bounded linearly by the number of vertices. Note that our reduction is different from the standard reduction of parity games to one-dimensional mean-payoff games where exponential weights are necessary [19]. A parity game consists of a finite-state game graph G along with a priority function $p : E \to \{1, \ldots, k\}$ that maps every edge to a natural number (the priority). The objective of player 1 is to ensure that the *minimal* priority that occurs infinitely often in a play is *even*, and the goal of player 2 is the complement.

The Reduction. Given a game graph G with priority function p we construct a multi-dimensional mean-payoff objective with weight function w of k dimensions on G as follows: for every $i \in \{1, \ldots, k\}$ we assign $w_i(e)$ as follows: (i) 0 if $p(e) > i$; (ii) -1 if $p(e) \leq i$ and $p(e)$ is odd; and (iii) n if $p(e) \leq i$ and $p(e)$ is even. From a vertex v, player 1 wins the parity game iff she wins the multidimensional mean-payoff game.

Theorem 2. *Let G be a game graph with a parity objective defined by a priority function of k-priorities. We can construct in linear time a k-dimensional weight function w, with maximal weight W bounded by n, such that a vertex v is winning for player 1 in the parity game iff v is winning for player 1 in the multidimensional mean-payoff game.*

3 Multidimensional Mean-Payoff Pushdown Graphs

We consider pushdown graphs (pushdown systems) with multidimensional mean-payoff objectives, and give an algorithm that determines if there exists a path that satisfies a multidimensional objective. Our algorithm runs in polynomial time even for arbitrary number of dimensions and for arbitrary weight function. We again use the hyperplane separation technique to reduce the problem to one-dimensional pushdown graphs, and a polynomial solution for the latter is known [12].

Key Obstacles and Overview of the Solution. We first describe the key obstacles for the polynomial time algorithm to solve pushdown graphs with multidimensional mean-payoff objectives (as compared to finite-state graphs and finite-state games). For pushdown graphs we need to overcome the next three main obstacles: (a) The mean-payoff value of a finite-state graph is uniquely determined by the weights of the simple cycles

of the graph. However, for pushdown graphs it is also possible to *pump* special types of acyclic paths. Hence, we first need to characterize the *pumpable paths* that uniquely determine the possible mean-payoff value vector in a pushdown graph. (b) Lemma 2 does not hold for arbitrary infinite-state graphs and we need to show that it does hold for pushdown graphs. (c) We require an algorithm to decide whether there is a hyperplane such that all the weights of the pumpable paths of a pushdown graph lie below the hyperplane (also for arbitrary dimensions). The overview of our solutions to the above obstacles are as follows: (a) In the first part of the section (until Proposition 1) we present a characterization of the pumpable paths in a pushdown graph. (b) We use Gordan's Lemma [15] (a special case of Farkas' Lemma) and in Lemma 6 we prove that Lemma 1 and Lemma 2 hold also for pushdown graphs (Lemma 1 holds for any infinite-state graph). (c) Conceptually, we find the separating hyperplane by constructing a matrix A, such that every row in A is a weight vector of a pumpable path, and we solve the linear inequality $\boldsymbol{\lambda} \cdot A < \mathbf{0}$. However, in general the matrix A can be of exponential size. Thus we need to use advanced linear-programing technique that solves in polynomial time linear inequalities with polynomial number of variables and exponential number of constraints. This technique requires a polynomial-time oracle that for a given $\boldsymbol{\lambda}$ returns a violated constraint (or says that all constraints are satisfied). We show that in our case the required oracle is the algorithm for pushdown graphs with one-dimensional mean-payoff objective (which we obtain from [12]), and thus we establish a polynomial-time hyperplane separation technique for pushdown graphs.

Stack Alphabet and Commands. We start with the basic notion of stack alphabet and commands. Let Γ denote a finite set of *stack alphabet*, and $\mathsf{Com}(\Gamma) = \{\mathit{skip}, \mathit{pop}\} \cup \{\mathit{push}(z) \mid z \in \Gamma\}$ denotes the set of *stack commands* over Γ. Intuitively, the command *skip* does nothing, *pop* deletes the top element of the stack, $\mathit{push}(z)$ puts z on the top of the stack. For a stack command com and a stack string $\alpha \in \Gamma^+$ we denote by $\mathit{com}(\alpha)$ the stack string obtained by executing the command com on α (in a stack string the top denotes the right end of the string).

Multi-weighted Pushdown Systems. A *multi-weighted pushdown system (WPS)* (or a multi-weighted pushdown graph) is a tuple: $\mathcal{A} = \langle Q, \Gamma, q_0 \in Q, E \subseteq (Q \times \Gamma) \times (Q \times \mathsf{Com}(\Gamma)), w : E \to \mathbb{Z}^k \rangle$, where Q is a finite set of *states* with q_0 as the initial state; Γ the finite *stack alphabet* and we assume there is a special initial stack symbol $\bot \in \Gamma$; E describes the set of edges or transitions of the pushdown system; and w is a weight function that assigns an integer weight vector to every edge; we denote by w_i the projection of w to the i-th dimension. We assume that $\bot$ can be neither put on nor removed from the stack. A *configuration* of a WPS is a pair (α, q) where $\alpha \in \Gamma^+$ is a stack string and $q \in Q$. For a stack string α we denote by $\mathsf{Top}(\alpha)$ the top symbol of the stack. The initial configuration of the WPS is $(\bot, q_0)$.

Successor Configurations and Runs. Given a WPS $\mathcal{A}$, a configuration $c_{i+1} = (\alpha_{i+1}, q_{i+1})$ is a *successor* configuration of a configuration $c_i = (\alpha_i, q_i)$, if there is an edge $(q_i, \gamma_i, q_{i+1}, \mathit{com}) \in E$ such that $\mathit{com}(\alpha_i) = \alpha_{i+1}$, where $\gamma_i = \mathsf{Top}(\alpha_i)$. A *path* π is a sequence of configurations. A path $\pi = \langle c_1, \ldots, c_{n+1} \rangle$ is a *valid path* if for all $1 \leq i \leq n$ the configuration c_{i+1} is a successor configuration of c_i (and the notation is similar for infinite paths). In the sequel we shall refer only to valid paths. A path can equivalently be defined as a sequence $\langle c_1 e_1 e_2 \ldots e_n \rangle$, where c_1 is the initial

configuration and e_i are valid transitions. We will present an algorithm that given a WPS $\mathcal{A}$ decides if there exists an infinite path π in $\mathcal{A}$ from q_0 such that $\mathsf{LimAvg}(\pi) \geq \mathbf{0}$.

Notations. We shall use (i) γ or γ_i for an element of Γ; (ii) e or e_i for a transition (equivalently an edge) from E; (iii) α or α_i for a string from Γ^*. For a path $\pi = \langle c_1, c_2, \ldots \rangle = \langle c_1 e_1 e_2 \ldots \rangle$ we denote by (i) q_i: the state of configuration c_i, and (ii) α_i: the stack string of configuration c_i.

Pumpable Pair of Paths. Let $\pi = \langle c_1 e_1 e_2 \ldots \rangle$ be a finite or infinite path. A *pumpable pair of paths* for π is a pair of non-empty sequences of edges: $(p_1, p_2) = (e_{i_1} e_{i_1+1} \ldots e_{i_1+n_1}, e_{i_2} e_{i_2+1} \ldots e_{i_2+n_2})$, for $n_1, n_2 \geq 0$, $i_1 \geq 0$ and $i_2 > i_1 + n_1$ such that for every $j \geq 0$ the path $\pi^j_{(p_1,p_2)}$ obtained by pumping the pair of paths p_1 and p_2 for j times each is a valid path.

Local Minimum of a Path. Let $\pi = \langle c_1, c_2, \ldots \rangle$ be a path. A configuration $c_i = (\alpha_i, q_i)$ is a *local minimum* if for every $j \geq i$ we have $\alpha_i \sqsubseteq \alpha_j$ (i.e., the stack string α_i is a prefix string of α_j). One basic fact: Every infinite path has infinitely many local minimum.

Non-Decreasing Paths and Cycles, and Proper Cycles. A path from configuration $(\alpha\gamma, q_1)$ to configuration $(\alpha\gamma\alpha_2, q_2)$ is a *non-decreasing α-path* if $(\alpha\gamma, q_1)$ is a local minimum. Note that if π is a non-decreasing α-path for some $\alpha \in \Gamma^*$, then the same sequence of transitions leads to a non-decreasing β-path for every $\beta \in \Gamma^*$. Hence we say that π is a non-decreasing path if there exists $\alpha \in \Gamma^*$ such that π is a non-decreasing α-path. A *non-decreasing cycle* is a non-decreasing path from (α_1, q) to (α_2, q) such that the top symbols of α_1 and α_2 are the same. A non-decreasing cycle from (α_1, q) to (α_2, q) is a *proper cycle* if $\alpha_1 = \alpha_2$ (i.e., returns to the same configuration). By convention, when we say that a path π is a non-decreasing path from (γ_1, q_1) to (γ_2, q_2), it means that for some $\alpha_1, \alpha_2 \in \Gamma^*$, the path π is a non-decreasing path from $(\alpha_1\gamma_1, q_1)$ to $(\alpha_1\gamma_1\alpha_2\gamma_2, q_2)$.

Cone of Pumpable Pairs. We denote $\mathbb{R}_+ = [0, +\infty)$. For a finite non-decreasing path π we denote by $\mathsf{PPS}(\pi)$ the (finite) set of pumpable pairs that occur in π, that is, $\mathsf{PPS}(\pi) = \{(p_1, p_2) \in (E^* \times E^*) \mid p_1 \text{ and } p_2 \text{ are a pumpable pair in } \pi\}$. Let $\mathsf{PPS}(\pi) = \{P_1 = (p_1^1, p_2^1), P_2 = (p_1^2, p_2^2), \ldots, P_j = (p_1^j, p_2^j)\}$, and we denote by $\mathsf{PumpMat}(\pi)$ the matrix that is formed by the weight vectors of the pumpable pairs of π, that is, the matrix has j rows and the i-th row of the matrix is $w(p_1^i) + w(p_2^i)$ (every weight vector is a row in the matrix). We denote by $\mathsf{PCone}(\pi)$ the cone of the weight vectors in $\mathsf{PPS}(\pi)$, formally, $\mathsf{PCone}(\pi) = \{\mathsf{PumpMat}(\pi) \cdot \boldsymbol{x} \mid \boldsymbol{x} \in (\mathbb{R}^k_+ \setminus \{\mathbf{0}\})\}$.

Fix $\ell = (|Q| \cdot |\Gamma|)^{(|Q| \cdot |\Gamma|)^2 + 1}$ for the rest of the section. For $q_1, q_2 \in Q$ and $\gamma_1, \gamma_2 \in \Gamma$, by abuse of notation we denote by $\mathsf{PPS}((\gamma_1, q_1), (\gamma_2, q_2))$ the (finite) set of all pumpable pair of paths, not longer than ℓ, that occur in a non-decreasing path from (γ_1, q_1) to (γ_2, q_2); we similarly define $\mathsf{PumpMat}((\gamma_1, q_1), (\gamma_2, q_2))$ and $\mathsf{PCone}((\gamma_1, q_1), (\gamma_2, q_2))$. If $q_1 = q_2$ and $\gamma_1 = \gamma_2$, then we abbreviate $\mathsf{PPS}((\gamma_1, q_1), (\gamma_1, q_1))$ by $\mathsf{PPS}((\gamma_1, q_1))$, and similarly for $\mathsf{PumpMat}$ and PCone. In Proposition 1 we establish a sufficient and necessary condition for the existence of a path with non-negative mean-payoff values in all the dimensions.

Proposition 1. *There exists an infinite path π such that $\mathsf{LimInfAvg}(\pi) \geq \mathbf{0}$ if and only if there exists a (reachable) non-decreasing cycle π such that $\mathbb{R}^k_+ \cap \mathsf{PCone}(\pi) \neq \emptyset$.*

By Proposition 1, we can decide whether there is an infinite path π for which $\mathsf{LimAvg}(\pi) \geq \mathbf{0}$ by checking if there exist a tuple $(\gamma, q) \in \Gamma \times Q$ for which there is a non-negative (and non-trivial) solution for the equation $\mathsf{PumpMat}((\gamma, q)) \cdot \boldsymbol{x} \geq 0$. As in Lemma 1 by adding k self-loop transitions with weights, where the weight of transition i is -1 in the i-th dimension and 0 in the other dimensions, we reduce the problem to finding q and γ such that there is a non-negative solution for $\mathsf{PumpMat}((\gamma, q)) \cdot \boldsymbol{x} = 0$. We present an algorithm that solves the problem by a reduction to a corresponding one dimensional problem. As before given a k-dimensional weight function w and a k-dimensional vector $\boldsymbol{\lambda}$ we denote by $w \cdot \boldsymbol{\lambda}$ the one-dimensional weight function obtained by multiplying the weight vectors with $\boldsymbol{\lambda}$. The reduction to one-dimensional objective (Lemma 6) requires the use of Gordan's lemma (Lemma 5).

Lemma 5 ([15]). *For a matrix A, either $A \cdot \boldsymbol{x} = \mathbf{0}$ has a non-trivial non-negative solution for $\boldsymbol{x}$, or there exists a vector $\boldsymbol{y}$ such that $\boldsymbol{y} \cdot A^T$ is negative in every dimension.*

Lemma 6. *Given a WPS $\mathcal{A}$ with a k-dimensional weight function w, and $(\gamma, q) \in \Gamma \times Q$, there exists a non-trivial non-negative solution for $\mathsf{PumpMat}((\gamma, q)) \cdot \boldsymbol{x} = 0$ if and only if for every $\boldsymbol{\lambda} \in \mathbb{R}^k$ there is a non-decreasing path from (γ, q) to (γ, q) that contains a pumpable pair $P = (p_1, p_2)$ such that $(w \cdot \boldsymbol{\lambda})(P) \geq 0$.*

Proposition 2. *There is a polynomial time algorithm that given WPS $\mathcal{A}$ with k-dimensional weight function w, $(\gamma, q) \in \Gamma \times Q$, a vector $\boldsymbol{\lambda} \in \mathbb{Q}^k$, and a rational number $r \in \mathbb{Q}$ decides if there exists a pumpable pair of paths P in a non-decreasing cyclic path that begins at (γ, q) in $\mathcal{A}$, with $\frac{(w \cdot \boldsymbol{\lambda})(P)}{|P|} > r$ and $|P| \leq \ell$, and if such pair exists, it returns $\frac{w(P)}{|P|}$.*

Intuitively, the algorithm (for Proposition 2) is based on the algorithm for solving WPSs with one-dimensional mean-payoff objectives. We now show how to use the result of the proposition and a result from linear programming to solve the problem.

Polynomial-Time Separating Oracle. Consider a linear program over n variables and exponentially many constraints in n. Given a polynomial time *separating oracle* that for every point in space returns in polynomial time whether the point is feasible, and if infeasible returns a violated constraint, the linear program can be solved in polynomial time using the ellipsoid method [16]. We use this fact to show the following result.

Proposition 3. *There exists a polynomial time algorithm that decides whether for a given state q and a stack alphabet symbol γ there exists a non-trivial non-negative solution for $\mathsf{PumpMat}((\gamma, q)) \cdot \boldsymbol{x} = 0$.*

Proof. Conceptually, given q and γ, we compute a matrix A, such that each row in A corresponds to the average weight vector of a row in $\mathsf{PumpMat}((\gamma, q))$ (that is, the weight of a pumpable pair divided by its length), and solves the following linear programming problem: For variables r and $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_k)$, the objective function is to minimize r subject to the constraints below: (i) $\boldsymbol{\lambda} \cdot A^T \leq \boldsymbol{r}$ where $\boldsymbol{r} = (r, r, \ldots, r)^T$; (ii) $\sum_{i=1}^{k} \lambda_i = 1$. Once the minimal r is computed, by Lemma 6, there exists a solution for $\mathsf{PumpMat}((\gamma, q)) \cdot \boldsymbol{x} = 0$ if and only if $r \geq 0$.

The number of rows of A in the worst case is exponential (to be precise at most $\ell \cdot (2 \cdot W \cdot \ell)^k$, since the length of the path is at most ℓ, the sum of weights is between $-W \cdot \ell$ and $W \cdot \ell$ and there are k dimensions). However, we do not enumerate the constraints

of the linear programming problem explicitly but use the result of linear programs with polynomial time separating oracle. By Proposition 2 we have an algorithm that verifies the feasibility of a solution (that is, an assignment for $\boldsymbol{\lambda}$ and r) and if the solution is infeasible it returns a constraint that is not satisfied by the solution. Thus the result of Proposition 2 provides the desired polynomial-time separating oracle. □

Theorem 3. *Given a WPS $\mathcal{A}$ with multidimensional weight function w, we can decide in polynomial time whether there exists a path π such that* $\mathsf{LimAvg}(\pi) \geq \mathbf{0}$.

4 Recursive Mean-Payoff Games with Modular Strategies

In this section, we will consider recursive games (which are equivalent to pushdown games) with modular strategies.

Weighted Recursive Game Graphs (WRGs). A *recursive game graph* $\mathcal{A}$ consists of a tuple $\langle A_0, A_1, \ldots, A_n \rangle$ of *game modules*, where each game module $A_i = (N_i, B_i, V_i^1, V_i^2, En_i, Ex_i, \delta_i)$ consists of the following components: (i) A finite nonempty set of *nodes* N_i. (ii) A nonempty set of *entry* nodes $En_i \subseteq N_i$ and a nonempty set of *exit* nodes $Ex_i \subseteq N_i$. (iii) A set of *boxes* B_i. (iv) Two disjoint sets V_i^1 and V_i^2 that partition the set of nodes and boxes into two sets, i.e., $V_i^1 \cup V_i^2 = N_i \cup B_i$ and $V_i^1 \cap V_i^2 = \emptyset$. The set V_i^1 (resp. V_i^2) denotes the places where it is the turn of player 1 (resp. player 2) to play (i.e., choose transitions). We denote the union of V_i^1 and V_i^2 by V_i. (v) A labeling $Y_i : B_i \to \{1, \ldots, n\}$ that assigns to every box an index of the game modules $A_1 \ldots A_n$. (vi) Let $\mathsf{Calls}_i = \{(b, e) \mid b \in B_i, e \in En_j, j = Y_i(b)\}$ denote the set of *calls* of module A_i and let $\mathsf{Retns}_i = \{(b, x) \mid b \in B_i, x \in Ex_j, j = Y_i(b)\}$ denote the set of *returns* in A_i. Then, $\delta_i \subseteq (N_i \cup \mathsf{Retns}_i) \times (N_i \cup \mathsf{Calls}_i)$ is the *transition relation* for module A_i. A *weighted recursive game graph* (for short WRG) is a recursive game graph, equipped with a weight function w on the transitions. We also refer the readers to [1] for detailed description and illustration with figures of recursive game graphs. We will also consider the special case of one-player WRGs, where either V^2 is empty (player-1 WRGs) or V^1 is empty (player-2 WRGs). The module A_0 is the initial module, and its entry node the starting node of the game.

Modular Strategies. Intuitively, a modular strategy only depends on the local history, and not on the context of invocation of the module. A *memoryless* modular strategy is defined in similar way, where every component local strategy is memoryless.

Mean-Payoff Objectives and Winning Modular Strategies. The *modular winning strategy problem* asks if player 1 has a modular strategy τ such that against every strategy σ for player 2 the play π given the starting node and the strategies satisfy $\mathsf{LimAvg}(\pi) \geq \mathbf{0}$ (note that the counter strategy of player 2 is a general strategy).

Undecidability. We show the following undecidability result, and in view of it will focus on WRGs under modular strategies for one-dimensional mean-payoff objectives.

Theorem 4. *The problem of deciding the existence of a modular winning strategy in WRGs with multidimensional mean-payoff objectives is undecidable, with six dimensions, three modules and with at most one exit for each module.*

NP-hardness. We consider WRGs under modular strategies with one-dimensional mean-payoff objectives. It was already shown in [12] that if the number of modules

is unbounded, then even if all modules have at most one exit, the problem is NP-hard even for one player game with weights restricted to $\{-1, 0, 1\}$. We present a similar hardness result by a reduction from 3SAT.

Theorem 5. *The decision problem of existence of modular winning strategies in WRG's with one-dimensional mean-payoff objectives is NP-hard even for WRG's with two modules and weights restricted to* $\{0, -1\}$.

Algorithm for One-Dimensional Mean-Payoff Objectives. Given the undecidability result, we focus on WRGs with one-dimensional mean-payoff objectives, and given the hardness results for either unbounded number of modules or unbounded number of exits, our goal is to present an algorithm that runs in polynomial time if both the number of modules and the number of exits are bounded. For the rest of this section we denote the number of game modules by M, the number of exits and boxes (in the entire graph) by E and B, respectively, and by n and m the maximal size of $|V_i|$ and $|\delta_i|$ (number of vertices and transitions) respectively that a module has. If M, E and W (the maximal absolute weight) are bounded, then our algorithm runs in polynomial time.

Description of the Key Steps of the Algorithm. One key result that we use is the fact that if there is a winning modular strategy, then there is a memoryless one [12]. The intuitive idea of our algorithm is to consider a *signature* function for a strategy that assigns to every exit of a module the weight of the *worst* sub-play from the entrance of the module to the exit. We show that for a memoryless modular winning strategy the range of the signature function is bounded (in absolute value) by $(n \cdot \mathsf{M})^{\mathsf{M}\cdot\mathsf{E}+1} \cdot W$. The final step is to show that for a given a signature function, there is a memoryless modular strategy to satisfy the signature function. The verification is achieved by solving a finite-state mean-payoff game. The detailed formal description is presented in [11].

Theorem 6. *Given a WRG $\mathcal{A}$ with a one-dimensional mean-payoff objective, whether player 1 has a modular winning strategy can be decided in* $(n \cdot \mathsf{M})^{O(\mathsf{M}^5+\mathsf{M}\cdot\mathsf{E}^2)} \cdot W^{O(\mathsf{M}^2+\mathsf{E})}$ *time.*

Hardness for Fixed Parameter Tractability. Given Theorem 6 (algorithm to solve in polynomial time when M and E are fixed) an interesting question is whether it is possible to show that WRGs under modular strategies is fixed parameter tractable (i.e., to obtain an algorithm that runs in time $O(f(\mathsf{M}, \mathsf{E}) \cdot \text{poly}(n, m, W))$). We show the hardness of fixed parameter tractability, again by a reduction from parity games, implying that fixed parameter tractability would imply the solution of the long-standing open problem of fixed parameter tractability of parity games (details presented in [11]).

Theorem 7. *Given a finite-state parity game G with n vertices and priority function of k-priorities, we can construct in polynomial time a WRG $\mathcal{A}$ with $2 \cdot k + 1$ modules, $O(k \cdot n)$ nodes, and weights in $\{-1, 0, +1\}$ such that a vertex v is winning for player 1 in the parity game iff there is a modular winning strategy in $\mathcal{A}$ with v as the initial node.*

References

1. Alur, R., La Torre, S., Madhusudan, P.: Modular strategies for recursive game graphs. Theor. Comput. Sci. 354(2), 230–249 (2006)

2. Beeri, C.: On the membership problem for functional and multivalued dependencies in relational databases. ACM Trans. on Database Systems 5, 241–259 (1980)
3. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 140–156. Springer, Heidelberg (2009)
4. Bohy, A., Bruyère, V., Filiot, E., Raskin, J.-F.: Synthesis from ltl specifications with mean-payoff objectives. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 169–184. Springer, Heidelberg (2013)
5. Boker, U., Chatterjee, K., Henzinger, T.A., Kupferman, O.: Temporal specifications with accumulative values. In: LICS, pp. 43–52 (2011)
6. Brázdil, T., Chatterjee, K., Kučera, A., Novotný, P.: Efficient controller synthesis for consumption games with multiple resource types. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 23–38. Springer, Heidelberg (2012)
7. Brim, L., Chaloupka, J., Doyen, L., Gentilini, R., Raskin, J.-F.: Faster algorithms for mean-payoff games. Formal Methods in System Design 38(2), 97–118 (2011)
8. Büchi, J.R., Landweber, L.H.: Solving sequential conditions by finite-state strategies. Transactions of the AMS 138, 295–311 (1969)
9. Chatterjee, K., Doyen, L., Henzinger, T.A.: Quantitative languages. ACM ToCL (2010)
10. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Generalized mean-payoff and energy games. In: FSTTCS, pp. 505–516 (2010)
11. Chatterjee, K., Velner, Y.: Finite-state and pushdown games with multi-dimensional mean-payoff objectives. CoRR, abs/1210.3141 (2012)
12. Chatterjee, K., Velner, Y.: Mean-payoff pushdown games. In: LICS (2012)
13. Droste, M., Meinecke, I.: Describing average- and longtime-behavior by weighted MSO logics. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 537–548. Springer, Heidelberg (2010)
14. Ehrenfeucht, A., Mycielski, J.: Positional strategies for mean payoff games. IJGT 8, 109–113 (1979)
15. Gordan, P.: Ueber die auflosung linearer gleichungen mit reellen coeffizienten. Mathematische Annalen 6, 23–28 (1873)
16. Grötschel, M., Lovász, L., Schrijver, A.: The ellipsoid method and its consequences in combinatorial optimization. Combinatorica 1(2), 169–197 (1981)
17. Gurvich, V.A., Karzanov, A.V., Khachiyan, L.G.: Cyclic games and an algorithm to find minimax cycle means in directed graphs. USSR Comp. Math. Phys. 28(5), 85–91 (1990)
18. Immerman, N.: Number of quantifiers is better than number of tape cells. Journal of Computer and System Sciences 22, 384–406 (1981)
19. Jurdzinski, M.: Deciding the winner in parity games is in UP ∩ co-UP. IPL 68 (1998)
20. Karp, R.M.: A characterization of the minimum cycle mean in a digraph. Discrete Mathematics 23, 309–311 (1978)
21. Kosaraju, S.R., Sullivan, G.F.: Detecting cycles in dynamic graphs in polynomial time. In: STOC, pp. 398–406 (1988)
22. Papadimitriou, C.H.: On the complexity of integer programming. JACM 28, 765–768 (1981)
23. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190. ACM Press (1989)
24. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete-event processes. SIAM Journal of Control and Optimization 25(1), 206–230 (1987)
25. 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)
26. Walukiewicz, I.: Pushdown processes: Games and model-checking. Inf. Comput. 164(2), 234–263 (2001)
27. Zwick, U., Paterson, M.: The complexity of mean payoff games on graphs. Theoretical Computer Science 158, 343–359 (1996)

Borel Determinacy of Concurrent Games

Julian Gutierrez[1] and Glynn Winskel[2]

[1] University of Oxford, Computer Science Department, Oxford, UK
[2] University of Cambridge, Computer Laboratory, Cambridge, UK

Abstract. Just as traditional games can be represented by trees, so concurrent games can be represented by event structures. We show the determinacy of such concurrent games with Borel sets of configurations as winning conditions, provided they are race-free and bounded-concurrent. Both properties are shown necessary. The determinacy proof proceeds via a reduction to the determinacy of tree games, and the determinacy of these in turn reduces to the determinacy of Gale-Stewart games.

1 Introduction

In logic the study of determinacy in games (the existence of a winning strategy or counter-strategy) dates back, at least, to Zermelo's work [12] on finite games which showed that all perfect-information finite games are determined. Since then, more complex games and determinacy results have been studied, *e.g.* for games with plays of infinite length. A research line that began in the 1950s with the seminal work of Gale and Stewart [5] on open games culminated with the work of Martin [6] who showed that two-player zero-sum sequential games with perfect information in which the winning conditions were Borel are determined.

In computer science determinacy results have most often been used rather than investigated. Frequently decision and verification problems are represented by games with winning conditions where winning strategies encode solutions to the problems being represented by the games. The determinacy of games ensures that in all cases there is a solution to the decision or verification problem under consideration, so is a computationally desirable property.

A common feature of the games mentioned above is that they are generally represented as trees. As a consequence, the plays of such games form total orders—the branches. The games we consider in this paper are not restricted to games represented by trees. Instead, they are played on games represented by event structures. Event structures [9] are the *concurrency* analogue of trees. Just as transitions systems unfold to trees, so Petri nets and asynchronous transition systems unfold to event structures. Plays are now partial orders of moves.

The concurrent games we consider are an extension of those introduced in [10]. Games there can be thought of as highly-interactive, distributed games between Player (thought of as a team of players) and Opponent (a team of opponents). The games model, as first introduced in [10], was extended with winning conditions in [3]. There a determinacy result was given for well-founded games

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 516–530, 2013.

(*i.e.* where only *finite* plays are possible) provided they are *race-free*, *i.e.* neither player could interfere with the moves available to the other—a property satisfied by all best-known games on trees/graphs, both sequential and concurrent.

Here we extend the main result of [3] by providing a much more general determinacy theorem. We consider concurrent games in which plays may be *infinite* and where the winning set of configurations forms a Borel set.

In particular we show that such games are determined provided that they are race-free and satisfy a structural condition we call *bounded concurrency*. Bounded concurrency expresses that no move of one of the players can be concurrent with infinitely many moves of the other—a condition trivially satisfied when *e.g.* all plays are finite, the games are sequential, or the games have rounds where simple choices are made (usual in traditional concurrent games). Bounded concurrency and race-freedom hold implicitly in games as traditionally defined.

We also show in what sense both bounded concurrency and race-freedom are necessary for Borel determinacy. Our determinacy proof follows by a reduction to the determinacy of Borel games, shown by Martin [6].

Related Work. Determinacy problems have been studied for more than a century: for finite games [12]; open games [5]; Borel games [6]; or Blackwell games [7], to mention a few particularly relevant to concurrency and computer science. Whereas the determinacy theorem in [3] is a concurrent generalisation of Zermelo's determinacy theorem for finite games, the determinacy theorem in this paper generalises the Borel determinacy theorem for infinite games from trees to event structures, so from total orders to partial orders of moves.

The results here apply to zero-sum concurrent games with perfect information. The games here require additional structure in order to model imperfect information [4] or stochastic features, so the determinacy result here does not apply directly to Blackwell games [7], the imperfect-information concurrent games played on graphs in [2] or the nonzero-sum concurrent games of [1].

Structure of the Paper. In Section 2 we present concurrent games represented as event structures. Section 3 introduces tree and Gale-Stewart games as variants of concurrent games. In Section 4 race-freedom and bounded concurrency are studied. Section 5 contains the determinacy theorem, preceding the conclusion.

2 Concurrent Games on Event Structures

An *event structure* comprises $(E, \leq, \mathrm{Con})$, consisting of a set E, of *events* which are partially ordered by $\leq$, the *causal dependency relation*, and a nonempty *consistency relation* Con consisting of finite subsets of E, which satisfy axioms:

$$\begin{array}{l}\{e' \mid e' \leq e\} \text{ is finite for all } e \in E,\\ \{e\} \in \mathrm{Con} \text{ for all } e \in E,\\ Y \subseteq X \in \mathrm{Con} \implies Y \in \mathrm{Con}, \text{ and}\\ X \in \mathrm{Con} \;\&\; e \leq e' \in X \implies X \cup \{e\} \in \mathrm{Con}.\end{array}$$

The *configurations* of E consist of those subsets $x \subseteq E$ which are

Consistent: $\forall X \subseteq x.\ X$ is finite $\Rightarrow X \in \mathrm{Con}$, and
Down-closed: $\forall e, e'.\ e' \leq e \in x \implies e' \in x$.

We write $\mathcal{C}^\infty(E)$ for the set of configurations of E and $\mathcal{C}(E)$ for the finite configurations. Two events e_1, e_2 which are both consistent and incomparable with respect to causal dependency in an event structure are regarded as *concurrent*, written $e_1 \mathrel{co} e_2$. In games the relation of *immediate* dependency $e \rightarrowtriangle e'$, meaning e and e' are distinct with $e \leq e'$ and no event in between plays an important role. For $X \subseteq E$ we write $[X]$ for $\{e \in E \mid \exists e' \in X.\ e \leq e'\}$, the down-closure of X; note if $X \in \mathrm{Con}$ then $[X] \in \mathrm{Con}$. We use $x \mathrel{-\!\!\subset} y$ to mean y covers x in $\mathcal{C}^\infty(E)$, *i.e.*, $x \subset y$ with nothing in between, and $x \overset{e}{\mathrel{-\!\!\subset}} y$ to mean $x \cup \{e\} = y$ for $x, y \in \mathcal{C}^\infty(E)$ and event $e \notin x$. We use $x \overset{e}{\mathrel{-\!\!\subset}}$, expressing that event e is enabled at configuration x, when $x \overset{e}{\mathrel{-\!\!\subset}} y$ for some configuration y.

Let E and E' be event structures. A *map* of event structures is a partial function on events $f : E \to E'$ such that for all $x \in \mathcal{C}(E)$ its direct image $fx \in \mathcal{C}(E')$ and if $e_1, e_2 \in x$ and $f(e_1) = f(e_2)$ (with both defined) then $e_1 = e_2$. The map expresses how the occurrence of an event e in E induces the coincident occurrence of the event $f(e)$ in E' whenever it is defined. Maps of event structures compose as partial functions, with identity maps given by identity functions. We say that the map is *total* if the function f is total. Say a total map of event structures is *rigid* when it preserves causal dependency.

The category of event structures is rich in useful constructions on processes. In particular, *pullbacks* are used to define the composition of *strategies*, while *restriction* (a form of equalizer) and the *defined part* of maps will be used in defining strategies. Any map of event structures $f : E \to E'$, which may be a partially defined on events, has a *defined part* the total map $f_0 : E_0 \to E'$, in which the event structure E_0 has events those of E at which f is defined, with causal dependency and consistency inherited from E, and where f_0 is simply f restricted to its domain of definition. Given an event structure E and a subset $R \subseteq E$ of its events, the *restriction* $E \restriction R$ is the event structure comprising events $\{e \in E \mid [e] \subseteq R\}$ with causal dependency and consistency inherited from E; we sometimes write $E \setminus S$ for $E \restriction (E \setminus S)$, where $S \subseteq E$.

Event Structures with Polarity. Both a game and a strategy in a game are represented with event structures with polarity, comprising an event structure E together with a polarity function $pol : E \to \{+, -\}$ ascribing a polarity $+$ (Player) or $-$ (Opponent) to its events; the events correspond to moves. Maps of event structures with polarity, are maps of event structures which preserve polarities. An event structure with polarity E is *deterministic* iff

$$\forall X \subseteq_{\mathrm{fin}} E.\ Neg[X] \in \mathrm{Con}_E \implies X \in \mathrm{Con}_E\,,$$

where $Neg[X] =_{\mathrm{def}} \{e' \in E \mid pol(e') = - \ \&\ \exists e \in X.\ e' \leq e\}$. We write $Pos[X]$ if $pol(e') = +$. The *dual*, $E^\perp$, of an event structure with polarity E comprises the same underlying event structure E but with a reversal of polarities.

Given two sets of events x and y, we write $x \subset^+ y$ to express that $x \subset y$ and $pol(y \setminus x) = \{+\}$; similarly, we write $x \subset^- y$ iff $x \subset y$ and $pol(y \setminus x) = \{-\}$.

Games and Strategies. Let A be an event structure with polarity—a game; its events stand for the possible moves of Player and Opponent and its causal dependency and consistency relations the constraints imposed by the game.

A *strategy (for Player)* in A is a total map $\sigma : S \to A$ from an event structure with polarity S, which is both *receptive* and *innocent*. Receptivity ensures an openness to all possible moves of Opponent. Innocence, on the other hand, restricts the behaviour of Player; Player may only introduce new relations of immediate causality of the form $\ominus \rightarrowtriangle \oplus$ beyond those imposed by the game.

Receptivity: A map σ is *receptive* iff
$\sigma x \overset{a}{\text{—}\!\subset}$ & $pol_A(a) = - \Rightarrow \exists! s \in S.\ x \overset{s}{\text{—}\!\subset}$ & $\sigma(s) = a$.

Innocence: A map σ is *innocent* iff
$s \rightarrowtriangle s'$ & ($pol(s) = +$ or $pol(s') = -$) then $\sigma(s) \rightarrowtriangle \sigma(s')$.

Say a strategy $\sigma : S \to A$ is *deterministic* if S is deterministic.

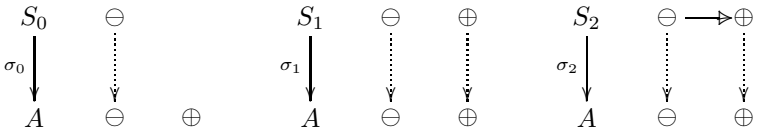
Composing Strategies. Suppose that $\sigma : S \to A$ is a strategy in a game A. A counter-strategy is a strategy of Opponent, so a strategy $\tau : T \to A^{\perp}$ in the dual game. The effect of playing-off a strategy σ against a counter-strategy τ is described via a pullback. Ignoring polarities, we have total maps of event structures $\sigma : S \to A$ and $\tau : T \to A$. Form their pullback,

$$\begin{array}{ccc} P & \xrightarrow{\Pi_2} & T \\ {\scriptstyle \Pi_1}\downarrow & \lrcorner & \downarrow{\scriptstyle \tau} \\ S & \xrightarrow[\sigma]{} & A\,. \end{array}$$

The event structure P describes the play resulting from playing-off σ against τ. Because σ or τ may be nondeterministic there can be more than one maximal configuration z in $\mathcal{C}^{\infty}(P)$. A maximal z images to a configuration $\sigma\Pi_1 z = \tau\Pi_2 z$ in $\mathcal{C}^{\infty}(A)$. Define the set of *results* of playing-off σ against τ to be

$$\langle \sigma, \tau \rangle =_{\text{def}} \{\sigma\Pi_1 z \mid z \text{ is maximal in } \mathcal{C}^{\infty}(P)\}\,.$$

Example 1. Let $\sigma_i : S_i \to A$ be a strategy in $A = \oplus\ co\ \ominus$



There are three analogous counter-strategies $\tau_j : T_j \to A^{\perp}$, $j = 0, 1, 2$, for Opponent. The results of playing each σ_i against each τ_j are:

$$\langle \sigma_i, \tau_j \rangle = \begin{cases} \{\emptyset\} & \text{if } i \in \{0,2\} \ \&\ j \in \{0,2\}, \\ \{\{\oplus\}\} & \text{if } i = 1 \ \&\ j = 0, \\ \{\{\ominus\}\} & \text{if } i = 0 \ \&\ j = 1, \\ \{\{\oplus, \ominus\}\} & \text{if } i = 1 \ \&\ j = 1. \end{cases}$$

Note that Player (or Opponent) can try to force *some* play to happen sequentially by adding causal dependencies, *e.g.* when using strategy σ_2 (or τ_2). This situation may lead to a deadlock as with σ_2 played-off against τ_2 when both players are waiting for their opponent to play first. □

Determinacy and Winning Conditions. A *game with winning conditions* [3] comprises $G = (A, W)$ where A is an event structure with polarity and the set $W \subseteq \mathcal{C}^\infty(A)$ consists of the *winning configurations (for Player)*. Define the *losing conditions (for Player)* to be $L = \mathcal{C}^\infty(A) \setminus W$. The dual $G^\perp$ of a game with winning conditions $G = (A, W)$ is defined to be $G^\perp = (A^\perp, L)$, a game where the roles of Player and Opponent are reversed, as are correspondingly the roles of winning and losing conditions.

A strategy in G is a strategy in A. A strategy in G is regarded as *winning* if it always prescribes moves for Player to end up in a winning configuration, no matter what the activity or inactivity of Opponent. Formally, a strategy $\sigma : S \to A$ in G is *winning (for Player)* if $\sigma x \in W$ for all $\oplus$-maximal configurations $x \in \mathcal{C}^\infty(S)$—a configuration x is $\oplus$-maximal if whenever $x \overset{s}{\text{—}\!\subset}$ then the event s has $-$ve polarity. Equivalently, a strategy σ for Player is winning if when played against any counter-strategy τ of Opponent, the final result is a win for Player; precisely, it can be shown [3] that a strategy σ is a winning for Player iff all the results $\langle \sigma, \tau \rangle$ lie within W, for any counter-strategy τ of Opponent. Sometimes we say a strategy σ *dominates* a counter-strategy τ (and *vice versa*) when $\langle \sigma, \tau \rangle \subseteq W$ (respectively, $\langle \sigma, \tau \rangle \subseteq L$). A game with winning conditions is *determined* when either Player or Opponent has a winning strategy in the game.

Example 2. Consider the game A with two inconsistent events $\oplus$ and $\ominus$ with the obvious polarities and winning conditions $W = \{\{\oplus\}\}$. The game (A, W) is not determined: no strategy of either player dominates all counter-strategies of the other player. Any strategy $\sigma : S \to A$ cannot be winning as it must by receptivity have $\ominus \mapsto \ominus$, so a $\oplus$-maximal configuration of S with image $\{\ominus\} \notin W$. By a symmetric argument no counter-strategy for Opponent can be winning. □

3 Tree Games and Gale–Stewart Games

We introduce tree games as a special case of concurrent games, traditional Gale–Stewart games as a variant, and show how to reduce the determinacy of tree games to that of Gale–Stewart games. Via Martin's theorem for the determinacy of Gale–Stewart games with Borel winning conditions we show that tree games with Borel winning conditions are determined.

3.1 Tree Games

Definition 3. Say E, an event structure with polarity, is *tree-like* iff it has empty concurrency relation (so $\leq_E$ forms a forest), all events enabled by the initial configuration $\emptyset$ have the same polarity, and is such that polarities alternate along branches, *i.e.* if $e \rightarrowtriangle e'$ then $pol_E(e) \neq pol_E(e')$. A *tree game* is (E, W), a concurrent game with winning conditions in which E is tree-like. □

Proposition 4. *Let E be a tree-like event structure with polarity. Then, its finite configurations $\mathcal{C}(E)$ form a tree w.r.t. $\subseteq$. Its root is the empty configuration $\emptyset$. Its (maximal) branches may be finite or infinite; finite sub-branches correspond to finite configurations of E; infinite branches correspond to infinite configurations of E. Its arcs, associated with $x \overset{e}{\multimap} x'$, are in 1-1 correspondence with events $e \in E$. The events e associated with initial arcs $\emptyset \overset{e}{\multimap} x$ all have the same polarity. In a branch*

$$\emptyset \overset{e_1}{\multimap} x_1 \overset{e_2}{\multimap} x_2 \overset{e_3}{\multimap} \cdots \overset{e_i}{\multimap} x_i \overset{e_{i+1}}{\multimap} \cdots$$

the polarities of the events $e_1, e_2, \ldots, e_i, \ldots$ alternate.

Proposition 4 gives the precise sense in which the terms 'arc,' 'sub-branch' and 'branch' are synonyms for the terms 'events,' 'configurations' and 'maximal configurations' when an event structure with polarity is tree-like. Notice that for a non-empty tree-like event structure with polarity, all the events that can occur initially share the same polarity. We say a non-empty tree game (E, W) has polarity $+$ or $-$ depending on whether its initial events are $+$ve (positive) or $-$ve (negative). We adopt the convention that the empty game $(\emptyset, \emptyset)$ has polarity $+$, and the empty game $(\emptyset, \{\emptyset\})$ has polarity $-$.

Proposition 5. *Let $f : S \to A$ be a total map of event structures with polarity and let A be tree-like. Then, it follows that S is also tree-like and that the map f is innocent. The map f is a strategy if and only if it is receptive.*

3.2 Gale–Stewart Games

Gale–Stewart games are a variant of tree games in which all maximal configurations of the tree game are infinite, and more importantly where Player and Opponent *must* play to a maximal, infinite configuration. Note that this is in general not the case for concurrent games where neither player is forced to play.

Definition 6. A *Gale–Stewart* game (G, V) comprises

- G, a tree-like event structure with polarity for which all maximal configurations are infinite, and
- V, a subset of infinite configurations—the *winning* configurations for Player.

A *winning strategy* in (G, V) is $\sigma : S \to G$, a deterministic strategy such that $\sigma x \in V$ for all maximal (and hence necessarily infinite) x in $\mathcal{C}^\infty(S)$. □

This is not the way a Gale–Stewart game and a winning strategy in a Gale–Stewart game are traditionally defined. However, because σ is deterministic it is injective as a map on configurations, so corresponds to the subfamily of configurations $T = \{\sigma x \mid x \in \mathcal{C}^\infty(S)\}$ of $\mathcal{C}^\infty(G)$. The family of configurations T forms a subtree of the tree of configurations of G. Its properties, given below, reconcile our definition based on event structures with the traditional one.

Proposition 7. *A winning strategy in a Gale–Stewart game (G, V) is a non-empty subset $T \subseteq \mathcal{C}^\infty(G)$ such that*

(i) $\forall x, y \in \mathcal{C}^\infty(G).\ \ y \subseteq x \in T \implies y \in T$,

(ii) $\forall x, y \in \mathcal{C}(G).\ \ x \in T\ \&\ x \overset{-}{\text{—}\subset} y \implies y \in T$,

(iii) $\forall x, y_1, y_2 \in T.\ x \overset{+}{\text{—}\subset} y_1\ \&\ x \overset{+}{\text{—}\subset} y_2 \implies y_1 = y_2$, *and*

(iv) all $\subseteq$-maximal members of T are infinite and in V.

A Gale–Stewart game (G, V) has a *dual* game $(G, V)^* =_{\text{def}} (G^\perp, V^*)$, where V^* is the set of all maximal configurations in $\mathcal{C}^\infty(G) \setminus V$. A winning strategy for Opponent in (G, V) is a winning strategy (for Player) in the dual game $(G, V)^*$.

For any event structure A there is a topology on $\mathcal{C}^\infty(A)$ given by the Scott open subsets [8]. The $\subseteq$-maximal configurations in $\mathcal{C}^\infty(A)$ inherit a sub-topology from that on $\mathcal{C}^\infty(A)$. The Borel subsets of a topological space comprise the sigma-algebra generated by the open subsets, *i.e.* the Borel sets are constructed by closing the open subsets under countable union, countable intersection and complement. Martin proved in [6] that Gale–Stewart games (G, V), with V Borel, are determined.

3.3 Determinacy of Tree Games

The determinacy of tree games with Borel winning conditions is shown by a reduction to the determinacy of Gale–Stewart games. Let (E, W) be a tree game. We construct a Gale–Stewart game $\mathrm{GS}(E, W) = (G, V)$ and a partial map $proj : G \to E$. The events of G are built as sequences of events in E together with two new symbols δ^- and δ^+ decreed to have polarity $-$ and $+$, respectively; the symbols δ^- and δ^+ represent delay moves by Opponent and Player.

An event of G is a non-empty finite sequence $[e_1, \cdots, e_k]$ of symbols from the set $E \cup \{\delta^-, \delta^+\}$ where: e_1 has the same polarity as (E, W); polarities alternate along the sequence; and for all subsequences $[e_1, \cdots, e_i]$, with $i \leq k$, $\{e_1, \cdots, e_i\} \cap E \in \mathcal{C}(E)$. Causal dependency is given by $[e_1, \cdots, e_k] \leq_G [e_1, \cdots, e_k, e_{k+1}]$ and consistency by compatibility w.r.t. $\leq_G$. Events $[e_1, \cdots, e_k]$ of G have the same polarity as their last entry e_k. Note that G is tree-like and that all maximal configurations are infinite (because of delay moves).

The map $proj : G \to E$ takes an event $[e_1, \cdots, e_k]$ of G to e_k if $e_k \in E$, and is undefined otherwise. The set V consists of all infinite, maximal configurations for which $proj\, x \in W$. We have built a Gale–Stewart game $\mathrm{GS}(E, W) = (G, V)$. Note, as *proj* is Scott-continuous on configurations, if W is Borel then so is V. The construction respects the duality on games: $\mathrm{GS}((E, W)^\perp) = (\mathrm{GS}(E, W))^*$.

Lemma 8. *Suppose σ is a winning strategy for $\mathrm{GS}(E, W)$. Then $proj \circ \sigma$ has defined part σ_0, a winning strategy for (E, W).*

Dually, a winning counter-strategy in $\mathrm{GS}(E, W)$ yields a winning counter-strategy in (E, W). Hence by Martin's Borel-determinacy theorem [6]:

Theorem 9. *Tree games with Borel winning conditions are determined.*

4 Race-Freedom and Bounded-Concurrency

Not all games are determined, *cf.* Example 2. However, a determinacy theorem holds for well-founded games (games where all configurations are finite) which satisfy a property called *race-freedom* (from [3]): in a game A, for $x \in \mathcal{C}(A)$,

$$x \xrightarrow{a} \subset \ \&\ x \xrightarrow{a'} \subset \ \&\ pol(a) \neq pol(a') \implies x \cup \{a, a'\} \in \mathcal{C}(A)\,. \quad (\mathbf{Race-free})$$

Note that the game in Example 2 is not *race-free*, but well-founded; tree games are race-free, but not necessarily well-founded. It may be easy to believe that a nondeterministic winning strategy always has a winning deterministic substrategy. This is not so and determinacy does not hold even for well-founded race-free games if we restrict to deterministic strategies, *cf.* [3]. Other observations made in [3]: being race-free is necessary for determinacy, in the sense that without it there are winning conditions for which a well-founded game is not determined; race-freedom is not sufficient to ensure determinacy when infinite behaviour is allowed, *i.e.* when A is not well-founded, as is illustrated in the following example.

Example 10. Let A be the event structure with polarity consisting of one positive event $\oplus$ which is concurrent with an infinite chain of alternating negative and positive events, *i.e.* for each i we have both $\oplus \mathrel{co} \oplus_i$ and $\oplus \mathrel{co} \ominus_i$, $i \in \mathbb{N}$,

$$A = \qquad \oplus \qquad \ominus_1 \longrightarrow \oplus_1 \longrightarrow \ominus_2 \longrightarrow \oplus_2 \longrightarrow \cdots$$

and Borel winning conditions (for Player) given by

$$W = \{\emptyset, \{\ominus_1, \oplus_1\}, ..., \{\ominus_1, \oplus_1, ..., \ominus_i, \oplus_i\}, ..., A\}.$$

So, Player wins if (i) no event is played, or (ii) the event $\oplus$ is not played and the play is finite and finishes in some $\oplus_i$, or (iii) all of the events in A are played. Otherwise, Opponent wins.

Player does not have a winning strategy because Opponent has an infinite family of *spoiler* strategies, not all be dominated by a single strategy of Player. The inclusion maps $\tau_\infty : T_\infty \to A^\perp$ and $\tau_i : T_i \to A^\perp$, $i \in \mathbb{N}$, are strategies for Opponent where $T_\infty^\perp =_{\text{def}} A$ and $T_i^\perp =_{\text{def}} A \setminus \{e' \in A \mid \ominus_i \leq e'\}$, for $i \in \mathbb{N}$.

Any strategy for Player that plays $\oplus$ is dominated by some strategy τ_i for Opponent; likewise, any strategy for Player that does not play $\oplus$ and plays only finitely many positive events $\oplus_i$ is also dominated by some strategy τ_i for Opponent. Moreover, a strategy for Player that does not play $\oplus$ and plays all of the events $\oplus_i$ in A is dominated by τ_∞. So, Player does not have a winning strategy in this game. Similarly, Opponent does not have a winning strategy in A because Player has two strategies that cannot be both dominated by any strategy for Opponent. Let $\sigma_{\overline{\oplus}} : S_{\overline{\oplus}} \to A$ and $\sigma_\oplus : S_\oplus \to A$ be strategies for Player such that $S_{\overline{\oplus}} =_{\text{def}} A \setminus \{\oplus\}$ and $S_\oplus =_{\text{def}} A$.

On the one hand, any strategy for Opponent that plays only finitely many (possibly zero) negative events $\ominus_i$ is dominated by $\sigma_{\overline{\oplus}}$; on the other, any strategy for Opponent that plays all of the negative events $\ominus_i$ in A is dominated by $\sigma_\oplus$. Thus neither player has a winning strategy in this game! □

In the above example, to win Player should only make the move $\oplus$ when Opponent has played a specified infinite number of moves. We can banish such difficulties by insisting that in a game A no event is concurrent with infinitely many events of the opposite polarity. This property is called *bounded-concurrency*:

$$\forall y \in \mathcal{C}^\infty(A).\ \forall a \in y.\ \{a' \in y \mid a \mathrel{co} a' \ \&\ pol(a) \neq pol(a')\} \text{ is finite.} \qquad (\mathbf{Bounded-concurrent})$$

Bounded concurrency is in fact a *necessary* structural condition for determinacy with respect to Borel winning conditions.

Notation. *For configurations y, y' of A, we shall write $max_+(y', y)$ iff y' is $\oplus$-maximal in y, i.e. $y' \overset{e}{\subset\!\!\!-} \subset$ & $pol(e) = + \Longrightarrow e \notin y$; similarly, $\overline{max}_+(y', y)$ iff y' is not $\oplus$-maximal in y. We use max_- analogously when $pol(e) = -$.* □

We show that if a *countable* race-free A is not bounded-concurrent, then there is Borel W so that the game (A, W) is not determined. Bounded-concurrency is thus shown necessary: Since A is not bounded-concurrent, there is $y \in \mathcal{C}^\infty(A)$ and $e \in y$ such that e is concurrent with infinitely many events of opposite polarity in y. W.l.o.g. assume that $pol(e) = +$, that $y_e =_{\text{def}} y \setminus \{e\}$ is a configuration and that $y = [e] \cup [\{a \in y \mid pol_A(a) = -\}]$. The following rules determine whether $y' \in \mathcal{C}^\infty(A)$ is in W or L:

1. $y' \supseteq y \Longrightarrow y' \in W$;
2. $y' \subset y$ & $e \in y' \Longrightarrow y' \in L$;
3. $y' \subset y$ & $e \notin y'$ & $max_+(y', y_e)$ & $\overline{max}_-(y', y_e) \Longrightarrow y' \in W$;
4. $y' \subset y$ & $e \notin y'$ & $\overline{max}_+(y', y_e)$ or $max_-(y', y_e) \Longrightarrow y' \in L$;
5. $y' \not\supseteq y$ & $(y' \cap y) \subset^- y' \Longrightarrow y' \in W$;
6. $y' \not\supseteq y$ & $(y' \cap y) \subset^+ y' \Longrightarrow y' \in L$;
7. otherwise assign y' (arbitrarily) to W.

No y' is assigned as winning for both Player and Opponent: the implications' antecedents are exhaustive and pairwise mutually exclusive.[1] Informally, rules 3 and 4 ensure that to win both players' strategies must progress towards y; rules 5 and 6 that to win no player can deviate from y; rules 1 and 2 that for Player to win they should make move e iff Opponent plays all their moves in y.

Lemma 11. *For A and W as above, W is a Borel subset of $\mathcal{C}^\infty(A)$ and the game (A, W) is not determined.*

Proof. (Sketch) Countability of A ensures that W defined using the scheme above is Borel. We first show: (i) if $\sigma : S \to A$ is a winning strategy for Player then y is σ-reachable, *i.e.*, there is $x \in \mathcal{C}^\infty(S)$ such that $\sigma x = y$—equivalently, there is τ such that $y \subseteq^+ y'$ for some $y' \in \langle \sigma, \tau \rangle$. And, (ii) if τ is a winning strategy for Opponent then y is τ-reachable.

Define the (deterministic) strategies $\tau_\infty : T_\infty \to A^\perp$ for Opponent and $\sigma_{\overline{\oplus}} : S_{\overline{\oplus}} \to A$ for Player as the following inclusion maps:

[1] The set W in Example 10 is an instance of this scheme—use rules 1 and 3.

$$\tau_\infty : A^\perp \restriction (\{a \in A \mid a \in y \text{ or } pol_A(a) = +\}) \hookrightarrow A^\perp,$$
$$\sigma_{\overline{\oplus}} : A \restriction (\{a \in A \mid a \in y_e \text{ or } pol_A(a) = -\}) \hookrightarrow A.$$

For (i) suppose $\sigma : S \to A$ is a winning strategy for Player. Let $y' \in \langle \sigma, \tau_\infty \rangle$. Thus $y' \in W$. Since $Neg[(\tau_\infty T_\infty)^\perp] \subseteq y$ then $(y' \cap y) \subset^- y'$ does not hold (to discard rule 5); and, because $\{-\} \not\subset pol_A(y' \setminus y)$ one can discard rule 7 too. Moreover, since $max_-(y', y_e)$ holds then $max_+(y', y_e)$ & $\overline{max}_-(y', y_e)$ does not hold (to discard 3). Then, necessarily $y' \supseteq y$ (by rule 1). However, because of the definition of τ_∞ this implies $y' \supseteq^+ y$ and that y is σ-reachable.

For (ii) suppose $\tau : T \to A^\perp$ is a winning strategy for Opponent. It is sufficient to show that y_e is τ-reachable as then y will also be τ-reachable by receptivity. Let $y' \in \langle \sigma_{\overline{\oplus}}, \tau \rangle$. Thus $y' \in L$. As $Pos[\sigma_{\overline{\oplus}} S_{\overline{\oplus}}] \subseteq y_e$ then $(y' \cap y) \subset^+ y'$ does not hold (to discard rule 6). Since there is no $s_e \in S_{\overline{\oplus}}$ such that $\sigma_{\overline{\oplus}}(s_e) = e$ then the antecedent of rule 2, *i.e.*, $y' \subset y$ & $e \in y'$, does not hold (to discard rule 2). And since $max_+(y', y_e)$ holds for all $y' \in \langle \sigma_{\overline{\oplus}}, \tau \rangle$ then, because $y' \in L$, we have that $max_-(y', y_e)$ holds too (by rule 4). Hence, y_e is τ-reachable.

To conclude we show there is no winning strategy for either player. If σ is a winning strategy for Player then by (i) there is $x \in \mathcal{C}^\infty(S)$ such that $\sigma x = y$; in particular there is $s_e \in x$ such that $\sigma(s_e) = e$. Define the inclusion map $\tau_{\text{fin}} : A^\perp \restriction (\{a \in A^\perp \mid a \in \sigma[s_e]_S \text{ or } pol_A(a) = +\} \hookrightarrow A^\perp$ as a spoiler strategy. Then τ_{fin} is a strategy for Opponent for which there is $y' \in \langle \sigma, \tau_{\text{fin}} \rangle$ with $e \in y'$ and where y' only contains finitely many $-$ve events. Either $y' \subset y$ whence $y' \in L$ by (2), or $y' \not\subset y$ whereupon $(y' \cap y) \subset^+ y'$ so $y' \in L$ by (6). Hence as τ_{fin} is a strategy for Opponent not dominated by σ the latter cannot be winning.

If τ is a winning strategy for Opponent then y is τ-reachable. Define a spoiler strategy as the inclusion map $\sigma_\oplus : A \restriction (\{a \in A \mid a \in y \text{ or } pol_A(a) = -\} \hookrightarrow A$. Then $\sigma_\oplus$ is a strategy for which there is $y' \in \langle \sigma_\oplus, \tau \rangle$ with $y' \supseteq y$. By (1), $y' \in W$, so $\sigma_\oplus$ is not dominated by τ, which then cannot be a winning strategy either. □

5 From Concurrent to Tree Games

We now construct a tree game $\text{TG}(A, W)$ from a concurrent game (A, W). We can think of the events of $\text{TG}(A, W)$ as corresponding to (non-empty) *rounds* of $-$ve (negative) or $+$ve (positive) events in the original concurrent game (A, W). When (A, W) is race-free and bounded-concurrent, a winning strategy for $\text{TG}(A, W)$ will induce a winning strategy for (A, W). In this way we reduce determinacy of concurrent games to determinacy of tree games.

5.1 The Tree Game of a Concurrent Game

Let (A, W) be a concurrent game; from the game (A, W) we construct a tree game $\text{TG}(A, W) = (TA, TW)$. The construction of TA depends on whether $\emptyset \in W$. When $\emptyset \in W$, define an alternating sequence of (A, W) to be a sequence

$$\emptyset \subset^- x_1 \subset^+ x_2 \subset^- \cdots \subset^+ x_{2i} \subset^- x_{2i+1} \subset^+ x_{2i+2} \subset^- \cdots$$

of configurations in $\mathcal{C}^\infty(A)$—the sequence need not be maximal. Define the $-$ve events of $\mathrm{TG}(A,W)$ to be $[\emptyset, x_1, x_2, \ldots, x_{2k-2}, x_{2k-1}]$, *i.e.* finite alternating sequences of the form $\emptyset \subset^- x_1 \subset^+ x_2 \subset^- \cdots \subset^+ x_{2k-2} \subset^- x_{2k-1}$, and let the $+$ve events to be $[\emptyset, x_1, x_2, \ldots, x_{2k-1}, x_{2k}]$, *i.e.* finite alternating sequences of the form $\emptyset \subset^- x_1 \subset^+ x_2 \subset^- \cdots \subset^- x_{2k-1} \subset^+ x_{2k}$, where $k \geq 1$. The causal dependency relation on TA is given by the relation of initial sub-sequence, with a finite subset of events being consistent if and only if the events are all initial sub-sequences of a common alternating sequence.

It is easy to see that a configuration of TA corresponds to an alternating sequence, the $-$ve events of TA matching arcs $x_{2k-2} \subset^- x_{2k-1}$ and the $+$ve events arcs $x_{2k-1} \subset^+ x_{2k}$. As such, we say a configuration $y \in \mathcal{C}^\infty(TA)$ is winning, and in TW, if and only if y corresponds to an alternating sequence of the form $\emptyset \cdots \subset^+ x_i \subset^- x_{i+1} \subset^+ \cdots$ for which $\bigcup_i x_i \in W$.

When $\emptyset \in L$, we define an alternating sequence of (A,W) as a sequence

$$\emptyset \subset^+ x_1 \subset^- x_2 \subset^+ \cdots \subset^- x_{2i} \subset^+ x_{2i+1} \subset^- x_{2i+2} \subset^+ \cdots$$

of configurations in $\mathcal{C}^\infty(A)$. In this case, the $-$ve events of $\mathrm{TG}(A,W)$ are finite alternating sequences ending in x_{2k}, while the $+$ve events end in x_{2k-1}, for $k \geq 1$. The remaining parts of the definition proceed analogously.

We have constructed a tree game $\mathrm{TG}(A,W)$ from (A,W). The construction respects duality on games: $\mathrm{TG}((A,W)^\perp) = (\mathrm{TG}(A,W))^\perp$.

Proposition 12. *Suppose (A,W) is a bounded-concurrent game. Every maximal alternating sequence has one of two forms,*

(i) finite:

$$\emptyset \cdots \subset^+ x_i \subset^- x_{i+1} \subset^+ \cdots x_k,$$

where x_i is finite for all $0 < i < k$ (where possibly x_k is infinite), or

(ii) infinite:

$$\emptyset \cdots \subset^+ x_i \subset^- x_{i+1} \subset^+ \cdots,$$

where each x_i is finite.

Example 13. Let (A,W) be the concurrent game with A as in Example 1 and $W = \{\emptyset, \{\oplus, \ominus\}\}$. Player has an obvious winning strategy: await Opponent's move and then make their move. Because $\emptyset \in W$, its tree game is

$$e_1 = [\emptyset, \{\ominus\}] \longrightarrow\!\!\!\!\!\rightarrow e_2 = [\emptyset, \{\ominus\}, \{\ominus, \oplus\}]$$

In the tree game the empty and maximal branches are winning. Its Gale–Stewart game has events which correspond to the non-empty subsequences of

$$(\delta^-\delta^+)^* e_1 (\delta^+\delta^-)^* e_2 (\delta^-\delta^+)^*$$

and branches which comprise consecutive sequences of such. An infinite branch is winning if it only has delay events or contains e_1 and e_2. Player has a winning strategy: delay while Opponent delays and play e_2 when Opponent plays e_1. □

5.2 Concurrent Strategies from Tree-Strategies

Now assume that the game (A, W) is race-free and bounded-concurrent. Suppose that $str : T \to TA$ is a (winning) strategy in the tree game $TG(A, W)$. Note that T is necessarily tree-like. We will show how to construct $\sigma_0 : S \to A$, a (winning) strategy in the original concurrent game (A, W). We construct S indirectly from $\mathcal{Q}$, a prime-algebraic domain [8,11], built as follows. For technical reasons, when defining $\mathcal{Q}$ it is convenient to assume that $A \cap (A \times T) = \emptyset$. Via *str* a sub-branch $\vec{t} = (t_1, \cdots, t_i, \cdots)$ of T determines a *tagged alternating sequence*

$$\emptyset \quad \cdots \quad \overset{t_{i-1}}{\subset^-} x_{i-1} \overset{t_i}{\subset^+} x_i \overset{t_{i+1}}{\subset^-} \cdots$$

where $str(t_i) = [\emptyset, \ldots, x_{i-1}, x_i]$. (the arc t_i is associated with a round extending x_{i-1} to x_i in the original game.) Define $q(\vec{t})$ to be the partial order with events

$$\bigcup \{(x_i \backslash x_{i-1}) \mid t_i \text{ is a } -\text{ve arc of } \vec{t}\} \ \cup \ \bigcup \{(x_i \backslash x_{i-1}) \times \{t_i\} \mid t_i \text{ is a } +\text{ve arc of } \vec{t}\}$$

—so a copy of the events $\bigcup_i x_i$ but with +ve events tagged by the +ve arc of T at which they occur—with order a copy of that $\bigcup_i x_i$ inherits from A with additional causal dependencies from (with x_{i-1}^- the set of $-$ve events in x_{i-1})

$$x_{i-1}^- \ \times \ ((x_i \setminus x_{i-1}) \times \{t_i\})$$

—making the +ve events occur after the $-$ve events which precede them in the alternating sequence.

Define the partial order $\mathcal{Q}$ as follows. Its elements are posets q, not necessarily finite, where for some sub-branch $(t_1, t_2, \cdots, t_i, \cdots)$ of T there is a *rigid inclusion* $q \hookrightarrow q(t_1, t_2, \cdots, t_i, \cdots)$, *i.e.* if $q(\vec{t}) \in \mathcal{Q}$ and $q \hookrightarrow q(\vec{t})$ is a rigid inclusion (regarded as a map of event structures) then $q \in \mathcal{Q}$. The order on $\mathcal{Q}$ is that of rigid inclusion. Define the function $\sigma : \mathcal{Q} \to \mathcal{C}^\infty(A)$ by taking

$$\sigma q = \{a \in A \mid a \text{ is } -\text{ve } \& \ a \in q\} \ \cup \ \{a \in A \mid \exists t \in T.\ a \text{ is } +\text{ve } \& \ (a, t) \in q\}$$

for $q \in \mathcal{Q}$. We check $\sigma q \in \mathcal{C}^\infty(A)$. Clearly, we have that $\sigma q(\vec{t}) = \bigcup_{i \in I} x_i$ where $\emptyset \quad \cdots \quad \overset{t_{i-1}}{\subset^-} x_{i-1} \overset{t_i}{\subset^+} x_i \overset{t_{i+1}}{\subset^-} \cdots$ is the tagged alternating sequence determined by $\vec{t} =_{\text{def}} (t_1, \cdots, t_i, \cdots)$. Any q for which there is a *rigid inclusion* $q \hookrightarrow q(\vec{t})$, *i.e.* which preserves causal dependency, is sent to a sub-configuration of $\bigcup_i x_i$.

Proposition 14. *Let $str : T \to TA$ be a strategy in the tree game $TG(A, W)$ and let $(t_1, \cdots, t_i, \cdots)$ be a sub-branch of T, so corresponding to some configuration $\{t_1, \cdots, t_i, \cdots\} \in \mathcal{C}^\infty(T)$. Then,*

$$str\{t_1, \cdots, t_i, \cdots\} \in TW \iff \sigma q(t_1, \cdots, t_i, \cdots) \in W\,.$$

The following proposition justifies writing $\subseteq$ for the order of $\mathcal{Q}$.

Proposition 15. *For all $q, q' \in \mathcal{Q}$, whenever there is an inclusion of the events of q in the events of q' there is a rigid inclusion $q \hookrightarrow q'$.*

The next lemma is crucial and depends critically on (A, W) being *race-free* and *bounded-concurrent*.

Lemma 16. *The order $(\mathcal{Q}, \subseteq)$ is a prime algebraic domain in which the primes are precisely those (necessarily finite) partial orders in $\mathcal{Q}$ with a top element.*

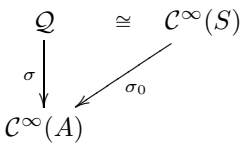
Proof. (Sketch) Any compatible finite subset X of $\mathcal{Q}$ has a least upper bound: if all the members of X include rigidly in a common q then taking the union of their images in q, with order inherited from q, provides their least upper bound. Provided $\mathcal{Q}$ has least upper bounds of directed subsets it will then be consistently complete with the additional property that every $q \in \mathcal{Q}$ is the least upper bound of the primes below it—this will make $\mathcal{Q}$ a prime algebraic domain. It then remains to show that $\mathcal{Q}$ has least upper bounds of directed sets.

Let S be a directed subset of $\mathcal{Q}$. The +ve events of orders $q \in S$ are tagged by +ve arcs of T. As S is directed the +ve tags which appear throughout all $q \in S$ must determine a common sub-branch of T, *viz.*, $\vec{t} =_{\text{def}} (t_1, t_2, \cdots, t_i, \cdots)$. Every +ve arc of the sub-branch appears in some $q \in S$ and all $-$ve arcs are present only by virtue of preceding a +ve arc. Forming the partial order $\bigcup S$ comprising the union of the events of all $q \in S$ with order the restriction of that on $q(\vec{t})$ we obtain a rigid inclusion $\bigcup S \hookrightarrow q(\vec{t})$ and so a least upper bound of S in $\mathcal{Q}$—from which prime algebraicity follows. □

Prime algebraic domains determine event structures in a simple way [8,11]: define S to be the event structure with polarity, with events the primes of $\mathcal{Q}$; causal dependency the restriction of the order on $\mathcal{Q}$; with a finite subset of events consistent if they include rigidly in a common element of $\mathcal{Q}$. The polarity of events of S is the polarity in A of its top element (the event is a prime in $\mathcal{Q}$).

Define $\sigma_0 : S \to A$ to be the function which takes a prime with top element an untagged event $a \in A$ to a and top element a tagged event (a, t) to a.

Lemma 17. *The function which takes $q \in \mathcal{Q}$ to the set of primes below q in $\mathcal{Q}$ gives an order isomorphism $\mathcal{Q} \cong \mathcal{C}^\infty(S)$. The function $\sigma_0 : S \to A$ is a strategy for which the following commutes:*



We obtain a winning strategy in a concurrent game from a winning strategy in its tree game:

Theorem 18. *Suppose that $str : T \to TA$ is a winning strategy in the tree game* $\mathrm{TG}(A, W)$. *Then $\sigma_0 : S \to A$ is a winning strategy in (A, W).*

Proof. (Sketch) For σ_0 to be a winning strategy we require that $\sigma_0 x \in W$ for every $\oplus$-maximal $x \in \mathcal{C}^\infty(S)$. Via the order isomorphism $\mathcal{Q} \cong \mathcal{C}^\infty(S)$ (Lemma 17)

we can carry out the proof in $\mathcal{Q}$ rather than $\mathcal{C}^\infty(S)$. For any q which is $\oplus$-maximal in $\mathcal{Q}$ (*i.e.* whenever $q \subseteq^+ q'$ in $\mathcal{Q}$ then $q = q'$) we require that $\sigma q \in W$.

Letting q be $\oplus$-maximal in $\mathcal{Q}$, because there is a rigid inclusion $q \hookrightarrow q(\vec{t})$ for some sub-branch $\vec{t} = (t_1, \cdots, t_i, \cdots)$ of T, we can show that $q = q(\vec{u})$ for some $\oplus$-maximal branch $\vec{u}$ of T. This implies that its image $str\{\vec{u}\}$ is in TW, as str is a winning strategy in $\mathrm{TG}(A, W)$. By Proposition 14, we have that $str\{\vec{u}\} \in TW \iff \sigma q(\vec{u}) \in W$. Hence, $\sigma q \in W$, as required. □

Corollary 19. *Let (A, W) be a race-free, bounded-concurrent game. If the tree game $\mathrm{TG}(A, W)$ has a winning strategy, then (A, W) has a winning strategy.*

As TG respects duality, a winning counter-strategy for $\mathrm{TG}(A, W)$ determines a winning counter-strategy for (A, W). Corollary 19 and Theorem 9 guarantee winning strategies in (A, W) from winning strategies in $\mathrm{GS}(\mathrm{TG}(A, W))$. We can now establish a *concurrent* analogue of Martin's determinacy theorem [6].

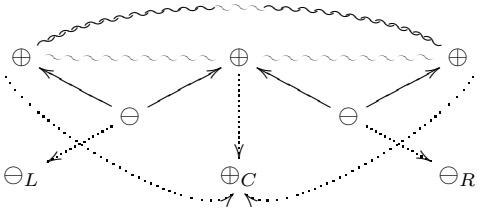
Theorem 20 (Concurrent Borel determinacy). *Any race-free, bounded-concurrent game (A, W), in which W is a Borel subset of $\mathcal{C}^\infty(A)$, is determined.*

We illustrate the construction of Theorem 18, how a winning strategy for a concurrent game is built from that of its tree game.

Example 21. Let (A, W) be a concurrent game where A is $\ominus_L co \oplus_C co \ominus_R$ and the set $\{\emptyset, \{\ominus_L, \oplus_C\}, \{\ominus_R, \oplus_C\}, \{\ominus_L, \ominus_R, \oplus_C\}\}$ is W, that is, Player's winning conditions in A. Player has a winning strategy. The maximal alternating sequences upon which the tree game $\mathrm{TG}(A, W)$ is constructed are:

1. $t^1_{max} = \emptyset \subset^- \{\ominus_L\} \subset^+ \{\ominus_L, \oplus_C\} \subset^- \{\ominus_L, \oplus_C, \ominus_R\}$,
2. $t^2_{max} = \emptyset \subset^- \{\ominus_R\} \subset^+ \{\ominus_R, \oplus_C\} \subset^- \{\ominus_R, \oplus_C, \ominus_L\}$,
3. $t^3_{max} = \emptyset \subset^- \{\ominus_L, \ominus_R\} \subset^+ \{\ominus_L, \ominus_R, \oplus_C\}$.

Its winning configurations correspond to those sub-branches terminating in W. It has a winning strategy str is given by the identity function on $\mathrm{TG}(A, W)$. We construct a winning strategy via a prime algebraic domain $\mathcal{Q}$ which has elements partial orders built out of tagged alternating sequences determined by str. In this example str is deterministic so the tagging plays no essential role and we can build the partial orders in $\mathcal{Q}$ from the alternating sequences above. The three maximal alternating sequences above are associated with the following partial orders on the three events $\{\ominus_L, \oplus_C, \ominus_R\}$: the first with just $\ominus_L \rightarrowtriangle \oplus_C$; the second with $\ominus_R \rightarrowtriangle \oplus_C$; and the last with both $\ominus_L \rightarrowtriangle \oplus_C$ and $\ominus_R \rightarrowtriangle \oplus_C$. There are other partial orders in $\mathcal{Q}$ associated with sub-branches. The event structure S of the winning non-deterministic concurrent strategy $\sigma_0 : S \to A$ is built from the complete primes of $\mathcal{Q}$, and takes the form shown:



Wiggly lines denote conflict and the dotted arrows the map σ_0. □

6 Concluding Remarks

Event structures have a central status within models for concurrency, both 'interleaving' and 'partial-order' based, and are formally related to other models by adjunctions. One can expect this central status to be inherited within games. Indeed, working with such a detailed model exposes new structure and new subtleties, which readily appear when studying determinacy issues.

For instance, in traditional 'interleaving' games on graphs or trees, both race-freedom and bounded-concurrency hold implicitly. At each vertex every player makes a simple choice *independently* of the others, implying race-freedom. Strategies are generally defined as functions from partial plays to partial plays via *rounds* which ensures bounded-concurrency. Round-free asynchrony, not studied before, makes the determinacy problem considerably harder.

Our determinacy result is, in a sense, the *strongest* one can hope to obtain (with respect to the descriptive complexity of the winning sets) for concurrent games on event structures—and hence on partial orders—since any generalisation of the winning sets would require an extension of the determinacy theorem by Martin [6]—well known to be at the limits of traditional set theory.

Acknowledgment. The authors acknowledge with gratitude the support of the ERC Advanced Grants RACE (at Oxford) and ECSYM (at Cambridge).

References

1. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. Journal of the ACM 49(5), 672–713 (2002)
2. Chatterjee, K., Henzinger, T.A.: A survey of stochastic ω-regular games. Journal of Computer and System Sciences 78(2), 394–413 (2012)
3. Clairambault, P., Gutierrez, J., Winskel, G.: The winning ways of concurrent games. In: LICS, pp. 235–244. IEEE Computer Society (2012)
4. Clairambault, P., Gutierrez, J., Winskel, G.: Imperfect information in logic and concurrent games. In: Coecke, B., Ong, L., Panangaden, P. (eds.) Abramsky Festschrift. LNCS, vol. 7860, pp. 7–20. Springer, Heidelberg (2013)
5. Gale, D., Stewart, F.: Infinite games with perfect information. Annals of Mathematical Studies 28, 245–266 (1953)
6. Martin, D.: Borel determinacy. Annals of Mathematics 102(2), 363–371 (1975)
7. Martin, D.: The determinacy of Blackwell games. Journal of Symbolic Logic 63(4), 1565–1581 (1998)
8. Nielsen, M., Plotkin, G., Winskel, G.: Petri nets, event structures and domains. Theoretical Computer Science 13, 85–108 (1981)
9. Nielsen, M., Winskel, G.: Models for concurrency. In: Handbook of Logic in Computer Science, pp. 1–148. Oxford University Press (1995)
10. Rideau, S., Winskel, G.: Concurrent strategies. In: LICS, pp. 409–418 (2011)
11. Winskel, G.: Prime algebraicity. Theoretical Computer Science 410(41), 4160–4168 (2009)
12. Zermelo, E.: On an application of set theory to the theory of the game of chess. In: Congress of Mathematicians, pp. 501–504. Cambridge University Press (1913)

A Faster Algorithm for Solving One-Clock Priced Timed Games*

Thomas Dueholm Hansen, Rasmus Ibsen-Jensen, and Peter Bro Miltersen

Department of Computer Science, Aarhus University, Denmark
{tdh,rij,pbmiltersen}@cs.au.dk

Abstract. One-clock priced timed games is a class of two-player, zero-sum, continuous-time games that was defined and thoroughly studied in previous works. We show that one-clock priced timed games can be solved in time $m12^n n^{O(1)}$, where n is the number of states and m is the number of actions. The best previously known time bound for solving one-clock priced timed games was $2^{O(n^2+m)}$, due to Rutkowski. For our improvement, we introduce and study a new algorithm for solving one-clock priced timed games, based on the sweep-line technique from computational geometry and the strategy iteration paradigm from the algorithmic theory of Markov decision processes. As a corollary, we also improve the analysis of previous algorithms due to Bouyer, Cassez, Fleury, and Larsen; and Alur, Bernadsky, and Madhusudan.

1 Introduction

Priced timed *automata* and priced timed *games* are classes of one-player and two-player zero-sum real-time games played on finite graphs that were defined and thoroughly studied in previous works [2,4,3,16,1,6,8,5,7,11,14]. Synthesizing (near-)optimal strategies for priced timed games has many practical applications in embedded systems design; we refer to the cited papers for references.

Informal Description of Priced Timed Games. Informally (for formal definitions, see the sections below), a priced timed game is played by two players on a finite directed labeled multi-graph. The vertices of the graph are called *states*, with some states belonging to Player 1 (or the Minimizer) and the other states belonging to Player 2 (or the Maximizer). We shall denote by n the total number of states of the game under consideration and m the total number of arcs (actions). Player 1 is trying to play the game to termination as cheaply as

* Work supported by the Sino-Danish Center for the Theory of Interactive Computation, funded by the Danish National Research Foundation and the National Science Foundation of China (under the grant 61061130540). The authors acknowledge support from the Center for research in the Foundations of Electronic Markets (CFEM), supported by the Danish Strategic Research Council. Thomas Dueholm Hansen is a recipient of the Google Europe Fellowship in Game Theory, and this research is supported in part by this Google Fellowship; he was also supported in part by The Danish Council for Independent Research | Natural Sciences (grant no. 12-126512).

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 531–545, 2013.

possible, while Player 2 is trying to make Player 1 pay as dearly as possible for playing. At any point in time, some particular state is the *current* state. The player controlling the state decides when to leave the current state and which arc to follow when doing so. For each arc, there is an associated *cost*. Each state has an associated *rate* of expense per time unit associated with waiting in the state. The above setup is further refined by the introduction of a finite number of *clocks* that can informally be thought of as "stop watches". In particular, some arcs may have associated a *reset* event for a clock. If the corresponding transition is taken, that clock is reset to 0. Also, an arc may have an associated clock and time interval. When the arm of the clock is in the interval, the corresponding transition can be taken; otherwise it can not. With three or more clocks, the problem of solving priced timed games is known not to be computable [5]. In this paper, we focus on the computable case of solving one-clock priced timed games. We shall refer to these as 1PTGs. We shall furthermore single out an important, particularly clean, special case of 1PTGs. We shall refer to this class as *simple priced timed games*, SPTGs. In an SPTG, time runs from 0 to 1, the single clock is never reset, and there are no restrictions on when transitions may be taken. A slightly more general class of games was called "[0,1]-PTGs without resets" by Bouyer et al. [7].

Values and Strategies. As is the case in general for two-player zero-sum games, informally, a priced timed game is said to have a *value* v if Player 1 and Player 2 are both able to guarantee (or approximate arbitrarily well) a total cost of v when the game is played. The guarantees are obtained when players commit to (near-)optimal *strategies* when playing the game. Player 1, who is trying to minimize cost, may (approximately) guarantee the value from above, while Player 2, who is trying to maximize cost, may (approximately) guarantee the value from below. Clearly, in general, the value of a one-clock priced timed game will be a function $v(q,t)$ of the initial state q and the initial setting t of the single clock. Bouyer *et al.* [7] showed that the value $v(q,t)$ exists[1] and that for any state q, the value function $t \to v(q,t)$ is a piecewise linear function of t. By *solving* a game, we mean computing an explicit description of all these functions (i.e., lists of their line segments). From such an object, near-optimal strategies can be synthesized.

Example. Figure 1 shows an SPTG with $n = 5$ states. Circles are controlled by Player 1 and squares are controlled by Player 2. States and actions have been annotated with rates and costs. If no cost is given for an action it has cost zero. The figure also includes graphs of the value functions. Actions are shown in black and gray, and an optimal strategy profile is shown along the x-axis of the value functions by using these colors – more precisely, it is the optimal strategy found by our algorithm. Waiting is shown as white. For instance, at state 2 at time $\frac{1}{3}$, Player 2 waits until time $\frac{2}{3}$ and then changes the current state to state 4.

[1] Players in general cannot guarantee the value exactly, but only approximate it arbitrarily well – one of the particular appealing aspects of SPTGs is that they *do* have exactly optimal strategies! This is in contrast to both the general case and [0,1]-PTGs without resets.

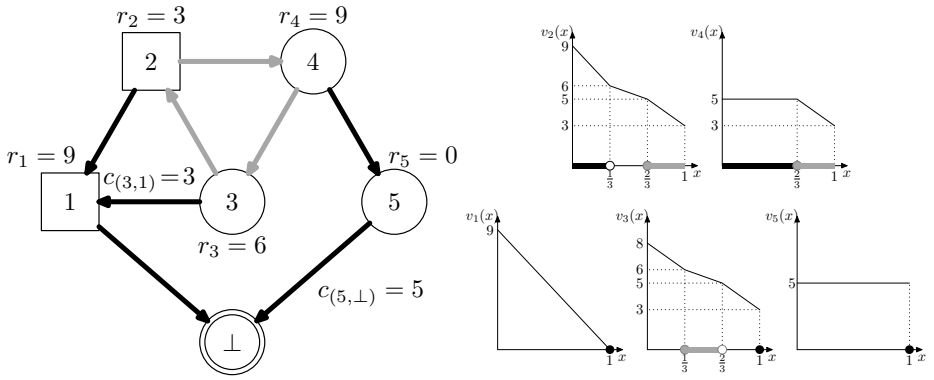


Fig. 1. Example of an SPTG, showing value functions and an optimal strategy profile

1.1 Contributions

The contributions of this paper are the following.

1. *A polynomial time Turing-reduction from the problem of solving general 1PTGs to the problem of solving SPTGs.* The best previous result[2] along these lines was a Turing-reduction from the general case to the case of "[0,1]-PTGs without resets" by Bouyer *et al.* [7]. Our reduction is a polynomial time reduction reducing solving a general 1PTG to solving at most $(n+1)(2m+1)$ SPTGs, while the previous reduction is an exponential time reduction.
2. *A novel algorithm for solving SPTGs, based on very different techniques than previously used to solve 1PTGs.* In particular, our algorithm is based on applications of a technique from computational geometry: the *sweep-line* technique of Shamos and Hoey [15], applied to the linear arrangement resulting when the graphs of all value functions are superimposed in a certain way. Also, an extension of Dijkstra's algorithm due to Khachiyan *et al.* [12] is a component of the algorithm. We believe that an implementation of this algorithm and the reduction could provide an attractive alternative to the current state-of-the-art tools for solving 1PTGs or various special cases (e.g., such as those of UPPAAL, `http://uppaal.org` or HyTech `http://embedded.eecs.berkeley.edu/ research/hytech/`), which all seem to be based on a value-iteration based algorithm independently devised by Bouyer, Cassez, Fleury, and Larsen [6]; and Alur, Bernadsky, and Madhusudan [1]. We shall refer to that algorithm as the BCFL-ABM algorithm.
3. *A worst case analysis of our algorithm as well as an improved worst case analysis of the BCFL-ABM algorithm.* Interestingly, the analysis of the

[2] Rutkowski [14] also made a polynomial time Turing-reduction, but his was for the decision problem: Given a state k in an 1PTG and a number v in unary, is the value of k at time 0 greater than v? Ours is for the similar decision problem, where v is given in binary.

algorithms is quite indirect: We analyze a different algorithm for a subproblem (priced games, see section 2), namely the *strategy iteration* algorithm, also used to solve Markov decision processes and various other classes of two-player zero-sum games played on graphs, and relate the analysis of this algorithm to our algorithm. To summarize the result of the analysis, it is convenient to introduce the parameter $L = L(G)$ of an SPTG to be the total number of distinct time coordinates of left endpoints of the linear segments of all value functions of G. Note that the parameter L is very natural, as L is a lower bound on the size of the explicit description of these value functions, i.e., the output of the algorithms under consideration. We show:

(a) For an SPTG G, we have that $L(G) \leq \min\{12^n, \prod_{k \in S}(|A_k|+1)\}$, where S is the set of states and A_k the set of actions in state k. The best previous bound on $L(G)$ was $2^{O(n^2)}$, due to Rutkowski [14]. It seems to be a "folklore theorem" that L does not become very big for games arising in practice. We conjecture that for all SPTGs G, $L(G) \leq p(n)$ for some polynomial p.

(b) The worst case time complexity of our new algorithm is $O((m+n\log n)L)$. In particular, the algorithm combined with the reduction solves general 1PTGs in time $m12^n n^{O(1)}$. The best previous worst case bound for any algorithm solving 1PTGs was $2^{O(n^2+m)}$, due to Rutkowski [14], who gave this bound for an alternative algorithm, due to him.

(c) The worst case number of iterations of the BCFL-ABM algorithm is $\min\{12^n, \prod_{k \in S}(|A_k|+1)\}m \cdot n^{O(1)}$ for general 1PTGs, significantly improving an analysis of Rutkowsi. (An "iteration" is a natural unit of time, specific to the algorithm – each iteration may take considerable time, as entire graphs of value functions are manipulated during an iteration).

The above bounds hold if we assume a unit-cost Real RAM model of computation, which is a natural model of computation for the algorithms considered (that previous analyses also seem to have implicitly assumed). The algorithms can also be analyzed in Boolean models of computation (such as the log cost integer RAM), as a rational valued input yields a rational valued output. Bounding the bit length of the numbers computed by straightforward inductive techniques, we find that this no more than squares the above worst case complexity bounds. The somewhat tedious analysis establishing this is not included in either version of the paper.

1.2 Organization of Paper

Our algorithm is most naturally presented in three stages, adding more complications to the model at each stage. First, in Section 2, we show how the strategy iteration paradigm can be used to solve *priced games*, where the temporal aspects of the games are not present. In Section 3, we show how the algorithm extends to simple priced timed games. In Section 4, we show how solving the general case of one-clock priced-timed games can be reduced to the case of simple priced timed games in polynomial time.

In terms of the list of contributions above, contribution 1) is Lemma 9. The algorithm of contribution 2) is `SolveSPTG` of Figure 3. Contribution 3a) is Theorem 4, contribution 3b) is Theorem 5 and contribution 3c) is Theorem 8.

2 Priced Games

In this section, we introduce *priced games*. A *priced game* G is given by a finite set of *states* $S = [n] = \{1, \ldots, n\}$, a finite set of *actions* $A = [m] = \{1, \ldots, m\}$. The set S is partitioned into S_1 and S_2, with S_i being the set of states belonging to Player i. Player 1 is also referred to as the *minimizer* and Player 2 is referred to as the *maximizer*. The set A is partitioned into $(A_k)_{k \in S}$, with A_k being the set of actions available in state k. Furthermore, define $A^i = \bigcup_{k \in S_i} A_k$. Each action $j \in A$ has an associated non-negative *cost* $c_j \in \mathbb{R}_{\geq 0} \cup \{\infty\}$ and an associated *destination* $d(j) \in S \cup \{\bot\}$, where $\bot$ is a special *terminal state*. Note that G can be interpreted as a directed weighted graph.

Positional Strategies and Payoffs. A *positional strategy* for Player i is a map σ_i of S_i to A, with $\sigma_i(k) \in A_k$ for each $k \in S_i$. A pair of strategies (or *strategy profile*) $\sigma = (\sigma_1, \sigma_2)$ defines a maximal *path* $P_{k_0,\sigma} = (k_0, k_1, \ldots)$, from each $k_0 \in S \cup \{\bot\}$, possibly ending at $\bot$, such that $d(\sigma(k_i)) = k_{i+1}$ for all $i \geq 0$. Note that σ can be naturally interpreted as a map from S to A. We let $\ell(k, \sigma)$ be the length of $P_{k,\sigma}$. The *payoff* $u(k, \sigma) \in \mathbb{R} \cup \{\infty\}$ of a state k for σ is defined to be the sum of the costs of actions on $P_{k_0,\sigma}$, or ∞ if $P_{k_0,\sigma}$ is of infinite length.

Values and Optimal Strategies. The *lower value* $\underline{v}(k)$ of a state k is defined by $\underline{v}(k) = \max_{\sigma_2} \min_{\sigma_1} u(k, \sigma_1, \sigma_2)$. A strategy σ_2 is called *optimal*, if for all states k, we have $\sigma_2 \in \operatorname{argmax}_{\sigma_2} \min_{\sigma_1} u(k, \sigma_1, \sigma_2)$. Similarly, the *upper value* $\overline{v}(k)$ of a state k is defined by $\overline{v}(k) = \min_{\sigma_1} \max_{\sigma_2} u(k, \sigma_1, \sigma_2)$ and a strategy σ_1 is called optimal if for all k, $\sigma_1 \in \operatorname{argmin}_{\sigma_1} \max_{\sigma_2} u(k, \sigma_1, \sigma_2)$. Khachiyan *et al.* [12] observed that $\underline{v}(k) = \overline{v}(k)$, i.e., that priced games have *values* $v(k) := \underline{v}(k) = \overline{v}(k)$. They also showed how to find these values and optimal strategies efficiently using a variant of Dijkstra's algorithm. The strategies found were postional, hence implying that optimal positional strategies always exists.

Strategy Iteration Algorithm. We shall present a different algorithm for solving priced games, following the general *strategy iteration* pattern [10]. This algorithm will be extended to simple priced timed games in the next section. Let σ be a strategy profile. For each state $k \in S$, we define the *valuation* $\nu(k, \sigma) = (u(k, \sigma), \ell(k, \sigma))$. I.e., the valuation of a state k for strategy profile σ is the payoff for k combined with the length of the path $P_{k,\sigma}$. We say that an action $j \in A_k$ from state k is an *improving switch for Player 1* if $(c_j + u(d(j), \sigma), 1 + \ell(d(j), \sigma)) < \nu(k, \sigma)$ where we order pairs lexicographically, with the first component being most significant. If the inequality is strict for the first component we say that j is a *strongly improving switch*. Improving switches are defined similarly for Player 2. The inclusion of the length in the definition of an improving switch is needed to ensure that the algorithm can find cycles of cost 0. We define $\sigma[j]$ to be the strategy profile obtained from σ by exchanging the action j with the corresponding action in σ.

The `StrategyIt` algorithm (see Figure 2) computes an optimal strategy profile σ and $u(\sigma)$, the vector of payoffs for σ, by repeatedly performing improving switches. The algorithm also works when multiple improving switches are performed in parallel. We refer to the full version of the paper [9] for proofs of the following lemma and theorem.

Lemma 1. *Let $\sigma = (\sigma_1, \sigma_2)$ be a strategy profile such that for both players i there are no improving switches in A^i. Then σ_1 and σ_2 are optimal.*

Theorem 1. *The* `StrategyIt` *algorithm correctly computes an optimal strategy profile σ^* such that neither player has an improving switch w.r.t. σ^*.*

Function `StrategyIt`(G, σ)

while $\exists$ *improving switch* $j \in A^1$ *for Player 1 w.r.t.* σ **do**
  $\sigma \leftarrow \sigma[j]$;
  while $\exists$ *improving switch* $j \in A^2$ *for Player 2 w.r.t.* σ **do**
    $\sigma \leftarrow \sigma[j]$;
return $(u(\sigma), \sigma)$;

Fig. 2. The `StrategyIt` algorithm for solving priced games

3 Simple Priced Timed Games

A *simple priced timed game* (SPTG) $G = (S_1, S_2, (A_k)_{k \in S}, (c_j)_{j \in A}, d, r)$ is given by a priced game $G' = (S_1, S_2, (A_k)_{k \in S}, (c_j)_{j \in A}, d)$, where $S = S_1 \cup S_2$ and $A = \bigcup_{k \in S} A_k$, and for each state $i \in S$, an associated *rate* $r_i \in \mathbb{R}_{\geq 0}$. We assume that $A_k \neq \emptyset$ for all $k \in S$.

Playing an SPTG. A SPTG G is played as follows. A pebble is placed on some starting state k_0 and the clock is set to its starting time x_0. The pebble is then moved from state to state by the players. The current configuration of the game is described by a state and a time, forming a pair $(k, x) \in S \times [0, 1]$.

Assume that after t steps the pebble is on state $k_t \in S_i$, controlled by Player i, at time x_t, corresponding to the configuration (k_t, x_t). Player i now chooses the next action $j_t \in A_{k_t}$. Furthermore, the player also chooses a delay $\delta_t \geq 0$ such that $x_{t+1} = x_t + \delta_t \leq 1$. The pebble is moved to $d(j_t) = k_{t+1}$. The next configuration is then (k_{t+1}, x_{t+1}). We write $(k_t, x_t) \xrightarrow{j_t, \delta_t} (k_{t+1}, x_{t+1})$. The game ends if $k_{t+1} = \bot$.

Plays and Outcomes. A *play* of the game is a sequence of steps starting from some configuration (k_0, x_0). Let $\rho = (k_0, x_0) \xrightarrow{j_0, \delta_0} (k_1, x_1) \xrightarrow{j_1, \delta_1} \dots \xrightarrow{j_{t-1}, \delta_{t-1}} (k_t, x_t)$ be a finite play such that $k_t = \bot$. The outcome of the game, paid by Player 1 to Player 2, is then given by $\text{cost}(\rho) = \sum_{\ell=0}^{t-1} (\delta_\ell r_{k_\ell} + c_{j_\ell})$. I.e., for each unit of time spent waiting at a state k Player 1 pays the rate r_k to Player 2.

Furthermore, every time an action j is used, Player 1 pays the cost c_j to Player 2. If ρ is an infinite play the outcome is ∞, and we write $\text{cost}(\rho) = \infty$.

Positional Strategies. A (positional) strategy for Player i is a map $\pi_i : S_i \times [0,1] \to A \cup \{\lambda\}$, where λ is a special delay action. For every $k \in S_i$ and $x \in [0,1)$, if $\pi_i(k,x) = \lambda$ then we require that there exists a $\delta > 0$ such that for all $0 \leq \epsilon < \delta$, $\pi_i(k, x+\epsilon) = \lambda$. Let $\delta_{\pi_i}(k,x) = \inf\{x' - x \mid x \leq x' \leq 1, \pi_i(k,x') \neq \lambda\}$ be the delay before the pebble is moved when starting in state k at time x for some strategy π_i. More general types of strategies could be considered, but see Remark 1.

Playing According to a Strategy and Strategy Profiles. Player i is said to play according to π_i if, when the pebble is in state $k \in S_i$ at time $x \in [0,1]$, he waits until time $x' = x + \delta_{\pi_i}(k,x)$ and then moves according to $\pi_i(k,x')$. A *strategy profile* $\pi = (\pi_1, \pi_2)$ is a pair of strategies, one for each player. Let Π_i be the set of strategies for Player i, and let Π be the set of all strategy profiles. A strategy profile π is again interpreted as a map $\pi : S \times [0,1] \to A \cup \{\lambda\}$. Furthermore, we use $\pi(x)$ to refer to the decisions at a fixed time. I.e., $\pi(x) : S \to A \cup \{\lambda\}$ is the map defined by $(\pi(x))(k) = \pi(k,x)$.

Value Functions and Optimal Strategies. Let $\rho^{\pi}_{k,x}$ be the play starting from configuration (k,x) where the players play according to π. Define the *value function* for a strategy profile $\pi = (\pi_1, \pi_2)$ and state k as: $v_k^{\pi_1,\pi_2}(x) = \text{cost}(\rho^{\pi}_{k,x})$. For fixed strategies π_1 and π_2 for Player 1 and 2, define the *best response value functions* for Player 2 and 1, respectively, for a state k as:

$$v_k^{\pi_1}(x) = \sup_{\pi_2 \in \Pi_2} v_k^{\pi_1,\pi_2}(x) \qquad v_k^{\pi_2}(x) = \inf_{\pi_1 \in \Pi_1} v_k^{\pi_1,\pi_2}(x)$$

Lower and *upper value functions* are defined analogously to the way they were defined in Section 2. Bouyer *et al.* [7] showed that the lower and upper value functions are the same. In fact, this was shown for the more general class of *one-clock priced timed games* (1PTGs) studied in Section 4. Hence, we again define the value function for every state k as:

$$v_k(x) := \sup_{\pi_2 \in \Pi_2} v_k^{\pi_2}(x) = \inf_{\pi_1 \in \Pi_1} v_k^{\pi_1}(x)$$

Remark 1. Positional strategies are commonly defined more generally as maps from states and times to delays and actions, i.e., $\tau_i : S_i \times [0,1] \to [0,1] \times A$. In the full version [9], we show that our less general notion of strategies is sufficient for optimal play. Sometimes strategies are defined even more generally such that they can be history-dependent*. Positional strategies are, however, still sufficient for optimal play in this setting (see Bouyer* et al. *[7]), and to simplify our proofs we restrict our attention to positional strategies.*

Strategies Optimal from Some Time. A strategy $\pi_i \in \Pi_i$ is *optimal from time x* for Player i if for all $k \in S$ and $x' \in [x,1]$, we have $v_k^{\pi_i}(x') = v_k(x')$. Strategies are called *optimal* if they are optimal from time 0. Similarly, a strategy

π_i is a *best response* to another strategy π_{-i} from time x if for all $k \in S$ and $x' \in [x, 1]$, we have $v_k^{\pi_i, \pi_{-i}}(x') = v_k^{\pi_{-i}}(x')$.

Nash Equilibrium from Some Time. A strategy profile (π_1, π_2) is called a *Nash equilibrium from time* x if π_1 is a best response to π_2 from time x, and π_2 is a best response to π_1 from time x. As in the case of priced games, any equilibrium payoff of an SPTG is the value of the game. The exact statement is shown in Lemma 2.

Lemma 2. *If there exists a strategy profile* (π_1, π_2) *that is a Nash equilibrium from time* x*, then* $v_k(x') = v_k^{\pi_1, \pi_2}(x')$ *for all* $k \in S$ *and* $x' \in [x, 1]$*.*

The existence of optimal strategies and best replies is non-trivial. We are, however, later going to prove the following theorem, which, in particular, implies that inf and sup can be replaced by min and max in the definitions of value functions. The theorem was first established by Bouyer *et al.* [7] for general 1PTGs.

Theorem 2. *For any SPTG there exists an optimal strategy profile. Also, the value functions are continuous piecewise linear functions.*

Our proof will be algorithmic. Specifically, the algorithm `SolveSPTG` computes a value function of the desired kind. Furthermore, the proof of correctness of `SolveSPTG` (the proof of Theorem 5) also yields the existence of exactly optimal strategies.

We refer to the non-differentiable points of the value functions of G as *event points* of G. The number of distinct event points of G is an important parameter in the complexity of our algorithm for solving SPTGs. We denote by $L(G)$ the total number of event points, excluding $x = 1$.

3.1 Solving SPTGs

In order to solve an SPTG we make use of a technique similar to the *sweep-line* technique from computational geometry of Shamos and Hoey [15]. Informally, we construct the value functions by moving a sweep-line backwards from time 1 to time 0, and at each time computing the current values based on the later values. The approach is also similar to a technique known in game theory as *backward induction*. The parameter of the induction, the time, is a continuous parameter, however. The BCFL-ABM algorithm also applies backward induction, but there the parameter of induction is the number of transitions taken, i.e., a discrete parameter, leading to a *value iteration* algorithm.

Informal Description of the Algorithm. If π is a strategy profile that is optimal from time x in an SPTG G, we use π to construct a new strategy profile π' that is optimal from time $x' < x$. More precisely, for x' sufficiently close to x, we show that there exists a fixed optimal action (where "waiting" is viewed as an action) for all states for both players for every point in time in the interval $[x', x)$. The new strategy profile π' is then obtained from π by using these actions.

Starting from time x', once the players wait in some state k, they wait at least until time x because they use the same actions throughout the interval. This allows us to model the situation with a priced game where every state k is given an additional action λ_k corresponding to waiting for $y = x - x'$ units of time. Thus, the value of a state in the priced game is the same as the value of the corresponding state in G if the game starts at time $x' = x - y$, and if the first time a player waits he is forced to wait until time x. The formal development of the algorithm follows. The proofs of lemmas in this section can be found in the full version [9] of the paper.

The Game $G^{x,y}$. Let G be some SPTG. We denote the priced game described above by $G^{x,y}$. Formally, $G^{x,y}$ is identical to the priced game defining G, with the exception that every state k is given an additional action λ_k leading to the terminal state $\perp$ with cost $v_k(x) + yr_k$. We refer to actions λ_k, for $k \in S$, as *waiting actions*. We sometimes write $u(k, \sigma, G^{x,y})$ instead of $u(k, \sigma)$ to clarify which priced game $G^{x,y}$ we consider.

Let $x \in (0, 1]$ and $y \geq 0$. Let σ be a strategy profile for $G^{x,y}$, and let k_0 be a state. Consider the (maximal) path $P_{k_0,\sigma} = (k_0, k_1, \dots)$ that starts at k_0 and uses actions of σ. We let $r(k_0, \sigma)$ (or $r(k, \sigma, G)$ when G is not clear from the context) be the rate of the state from which a waiting action is used in $P_{k_0,\sigma}$. If no waiting action is used in $P_{k_0,\sigma}$ we let $r(k_0, \sigma)$ be zero. Note that $r(k_0, \sigma)$ does not depend on y. In particular, we always have: $u(k_0, \sigma, G^{x,y}) = u(k_0, \sigma, G^{x,0}) + yr(k_0, \sigma)$.

We will often let y be the infinitesimal ϵ, in which case we simply denote $G^{x,\epsilon}$ by G^x. Since ϵ is an infinitesimal, the payoffs of a strategy profile σ for G^x have two components. We then (informally) have $u(k_0, \sigma, G^x) = u(k_0, \sigma, G^{x,0}) + \epsilon r(k_0, \sigma)$. There are no infinitesimals in $G^{x,0}$, and, hence, the second component of the payoff $u(k_0, \sigma, G^x)$ is exactly $r(k_0, \sigma)$. For every $x \in (0, 1]$ we let $\sigma^x = (\sigma_1^x, \sigma_2^x)$ be a strategy profile for which neither player has an improving switch. I.e., by Lemma 1 σ^x is an optimal strategy profile for G^x. The existence of σ^x is guaranteed by the correctness of the `StrategyIt` algorithm.

Note that the only difference between G^x and $G^{x'}$, for $x \neq x'$, is the costs of the waiting actions λ_k. Hence, we may interpret a strategy profile σ for G^x as a strategy profile for $G^{x'}$. Slightly abusing notation, we will interpret actions chosen by σ as also being actions $\pi(x)$ for G, and the actions of $\pi(x)$ as forming a strategy profile for G^x.

Lemma 3. *The strategy profile σ^x is optimal for $G^{x,0}$ and $u(k, \sigma^x, G^{x,0}) = v_k(x)$.*

Lemma 4. *Let π be a strategy profile for G that is optimal from time x, and let $x' < x$. If $\pi(x'') = \sigma^x$ for all $x'' \in [x', x)$, then $v_k^\pi(x') = v_k(x) + (x - x')r(k, \sigma^x, G^x)$ for all $k \in S$.*

The Function `NextEventPoint`(G^x). For every action $j \in A$ and time $x \in (0, 1]$, define the function:

$$\begin{aligned} f_{j,x}(x'') &:= c_j + u(d(j), \sigma^x, G^{x,x-x''}) \\ &= c_j + u(d(j), \sigma^x, G^{x,0}) + (x - x'')r(d(j), \sigma^x, G^x) \ . \end{aligned}$$

I.e., $f_{j,x}(x'')$, for $j \in A_k$, is the payoff obtained in $G^{x,x-x''}$ by starting at state k, using action j, and then repeatedly using actions of σ^x. In particular, we have $u(k, \sigma^x, G^{x,x-x''}) = f_{\sigma^x(k),x}(x'')$, and $j \in A_k$, for $k \in S_1$, is a strongly improving switch for Player 1 w.r.t. σ^x if and only if $f_{j,x}(x'') < f_{\sigma^x(k),x}(x'')$. A similar observation can be made for Player 2. Note that $f_{j,x}(x'')$ defines a line in the plane.

We define $\texttt{NextEventPoint}(G^x)$ to be equal to the first intersection before x of two lines $f_{\sigma^x(k),x}(x'')$ and $f_{j,x}(x'')$, for $k \in S$ and $j \in A_k \setminus \sigma^x$. It is not difficult to check that $\texttt{NextEventPoint}(G^x)$ can be defined as:

$$\max \{0\} \cup \{x' \in [0, x) \mid \exists k \in S, j \in A_k : f_{j,x}(x') = f_{\pi(k),x}(x') \wedge f_{j,x}(x) \neq f_{\pi(k),x}(x)\}.$$

Lemma 5. *Let $x' = \texttt{NextEventPoint}(G^x)$, then σ^x is optimal for $G^{x,y}$, for all $y \in (0, x-x']$. Furthermore, neither player has a strongly improving switch w.r.t. σ^x for $G^{x,y}$.*

We are now ready to state the main technical lemma used to prove the correctness of our algorithm. The main idea of the proof is to show that the strategy profile π' defined in the lemma is a Nash equilibrium from time x'. It is shown that if some player can improve over π' after time x' then he has a strongly improving switch w.r.t. σ^x for $G^{x,y}$, for some $y \in (0, x - x']$, which contradicts Lemma 5. Lemma 4 is used to prove that the optimal values have the desired form.

Lemma 6. *Let $x' = \texttt{NextEventPoint}(G^x)$, and let $\pi = (\pi_1, \pi_2)$ be a strategy profile that is optimal from time x. Then the strategy profile $\pi' = (\pi'_1, \pi'_2)$, defined by:*

$$\pi'(k, x'') = \begin{cases} \sigma^x(k) & \text{if } x'' \in [x', x) \\ \pi(k, x'') & \text{otherwise} \end{cases}$$

is optimal from time x', and $v_k(x'') = v_k(x) + r(k, \sigma^x, G^x)(x - x'')$, for $x'' \in [x', x)$ and $k \in S$.

The Algorithm. Lemma 6 allows us to compute optimal strategies by backward induction once the values $v_k(1)$ at time 1 are known for all states $k \in S$. Finding $v_k(1)$ and corresponding optimal strategies from time 1 is, however, not difficult. Indeed, when $x = 1$ time does not increase further, and we simply solve the priced game G' that defines G. The resulting algorithm is shown in Figure 3. Note that the choice of first using the `ExtendedDijkstra` algorithm of Khachiyan *et al.* [12] and then the `StrategyIt` algorithm is to facilitate the analysis in Section 3.2. In fact, any algorithm for solving priced games could be used. By observing that `SolveSPTG` simply repeatedly applies Lemma 6 to construct optimal strategies by backward induction we get the following theorem.

Theorem 3. *If* `SolveSPTG` *terminates, it correctly computes the value function and optimal strategies for both players.*

Note that `SolveSPTG` resembles the sweep-line algorithm of Shamos and Hoey [15] for the line segment intersection problem. At every time x we have n ordered sets of line segments with an intersection within one set at the next event point $x' = \texttt{NextEventPoint}(G^x)$. When handling the event point, the order of the line segments is updated, and we move on to the next event point.

Function `SolveSPTG`(G)

$(v(1), (\pi_1(1), \pi_2(1))) \leftarrow \texttt{ExtendedDijkstra}(G')$;
$x \leftarrow 1$;
while $x > 0$ **do**
  $(u(k, \sigma^x, G^{x,0}) + \epsilon r(k, \sigma^x, G^x), (\sigma_1, \sigma_2)) \leftarrow \texttt{StrategyIt}(G^x, (\pi_1(x), \pi_2(x)))$;
  $x' \leftarrow \texttt{NextEventPoint}(G^x)$;
  forall $k \in S$ *and* $x'' \in [x', x)$ **do**
    $v_k(x'') \leftarrow v_k(x) + r(k, \sigma^x, G^x)(x - x'')$;
    $\pi_1(k, x'') \leftarrow \sigma_1(k)$;
    $\pi_2(k, x'') \leftarrow \sigma_2(k)$;
  $x \leftarrow x'$;
return $(v, (\pi_1, \pi_2))$;

Fig. 3. Algorithm for solving a simple priced timed game $G = (G', (r_k)_{k \in S})$

3.2 Bounding the Number of Event Points

Let G be an SPTG. Recall that the only difference between G^x and $G^{x'}$, for $x \neq x'$, are the costs of actions λ_k, for $k \in S$, if $v_k(x) \neq v_k(x')$. The actions available from each state are therefore the same, and a strategy profile σ for G^x can, thus, also be interpreted as a strategy profile for $G^{x'}$. To bound the number of event points we assign a potential to each strategy profile σ, such that the potential strictly decreases when one of the players performs a single improving switch. Furthermore, the potential is defined independently of the values $v_k(x)$. It then follows that the number of single improving switches performed by the `SolveSPTG` algorithm is at most the total number of strategy profiles for G^x. We further improve this bound to show that the number of event points is at most exponential in the number of states. This improves the previous bound by Rutkowski [14].

The Potential Matrix. Let n be the number of states of G, let N be the number of distinct rates, including rate 0 for the terminal state $\bot$. Assume that the distinct rates are ordered such that $r_1 < r_2 < \cdots < r_N$. Recall that $r(k, \sigma)$ is the rate of the waiting state reached from k in σ. Let $\text{count}(\sigma, i, \ell, r) = |\{k \in S_i \mid \ell(k, \sigma) = \ell \ \wedge \ r(k, \sigma) = r\}|$ be the number of states controlled by Player i at distance ℓ from $\bot$ in σ that reach a waiting state with rate r.

For every strategy profile σ for the priced games G^x, for $x \in (0,1]$, define the potential $P(\sigma) \in \mathbb{N}^{n \times N}$ as an integer matrix as follows: $P(\sigma)_{\ell,r} = \text{count}(\sigma, 2, \ell, r) - \text{count}(\sigma, 1, \ell, r)$. I.e., rows correspond to lengths, columns correspond to rates, and entries count the number of corresponding Player 2 controlled states minus the number of corresponding Player 1 controlled states.

Intuitively, at time 1 the rates are unimportant and Player 2, the maximizer, will prefer using actions of large cost. However, the closer we get to time 0 the more important the rates become, and Player 2 may want to switch to actions of lower cost that benefit more from higher rates. The trade-off between costs and rates is monotone over time, and this will allows us to show that the potential matrices change monotonically when going from time 1 to time 0. The situation is the opposite for Player 1, which is the reason that the roles of the two players are opposite of each other.

Ordering the Potential Matrices. We define a lexicographic ordering of potential matrices where, firstly, entries corresponding to lower rates are of higher importance. Secondly, entries corresponding to shorter lengths are more important. Formally, we write $P(\sigma) \prec P(\sigma')$ if and only if there exists ℓ and r such that $P(\sigma)_{\ell',r'} = P(\sigma')_{\ell',r'}$ for all $r' < r$ and $1 \leq \ell' \leq n$, and $P(\sigma)_{\ell',r} = P(\sigma')_{\ell',r}$ for all $\ell' < \ell$, and $P(\sigma)_{\ell,r} < P(\sigma')_{\ell,r}$.

We get the following lemma, whose proof has been deferred to the full version [9].

Lemma 7. *Let σ be a strategy profile that is optimal for $G^{x,0}$, for some $x \in (0,1]$. Let $j \in A^i$ be an improving switch for Player i w.r.t. σ in the priced game G^x. Then $P(\sigma[j]) \prec P(\sigma)$.*

Consider the finite sequence of strategy profiles generated by `StrategyIt` while running the algorithm `SolveSPTG`. When solving G^x, for some $x \in (0,1]$, the players repeatedly perform single improving switches. The resulting optimal strategy profile σ^x is then used as the starting point for solving the next priced game $G^{x'}$, for $x' = \texttt{NextEventPoint}(G^x)$. Using Lemma 7 we show that the strategy profiles observed while running the `SolveSPTG` algorithm have decreasing potential matrices. This allows us to bound the number of event points by the number of strategy profiles of G^x, giving us the following theorems. For additional details, and an argument for the 12^n bound, see the full version [9].

Theorem 4. *The total number of event points for any SPTG G with n states is $L(G) \leq \min\{12^n, \prod_{k \in S}(|A_k| + 1)\}$.*

Theorem 5. `SolveSPTG` *solves any SPTG G in time $O(m \cdot \min\{12^n, \prod_{k \in S}(|A_k| + 1)\})$ in the unit cost model, where n is the number of states and m is the number of actions. Alternatively, the variant of* `SolveSPTG` *that uses the* `ExtendedDijkstra` *algorithm of Khachiyan* et al. *[12] instead of* `StrategyIt` *solves G in time $O(L(G)(m + n \log n))$.*

Theorem 2 follows as a corollary of Theorem 5, since `SolveSPTG` is always guaranteed to compute optimal strategies, and the resulting value functions are continuous piecewise linear functions.

4 One-Clock Priced Timed Games

One-clock priced timed games (1PTGs) extend SPTGs in two ways. First, actions are associated with time intervals (*existence intervals*) during which they are available, and second, certain actions (*reset actions*) will cause the time to be reset to zero. Also, we do not require the time to run from zero to one. In this extended abstract we only give an informal description of 1PTGs and of our results and proofs related to 1PTGs. We refer to the full version of the paper [9] for a more extensive presentation of the subject. To simplify the lemmas below, we say that an (n, m, r, d)-1PTG is a 1PTG with n states, r of which are the destination of some reset action, m actions and d distinct endpoints of existence intervals. We let M be the maximum endpoint of any existence interval, i.e., after time M no actions are available and the game must end.

Plays, Outcomes, Positional Strategies, Values, and ϵ-Optimal Strategies. 1PTGs are played like SPTGs with the exception that using a reset action resets the time to zero and that the actions must be available when they are used. Since 1PTGs are simply priced timed games with only one clock we also refer to the description of priced timed games in Section 1. Plays, outcomes, positional strategies, value functions, best response and lower and upper value functions are defined analogously to SPTGs, except that time goes from 0 to M instead of from 0 to 1 and that actions must be available when used by a strategy. It should be pointed out, however, that optimal strategies do not always exist (see Bouyer *et al.* [7]). Instead, we say that a strategy is *ϵ-optimal* for Player i for $\epsilon \geq 0$ if, for all states $k \in S$ and all times $x \in [0, M]$, $|v_k^{\pi_i}(x) - v_k(x)| \leq \epsilon$.

In the proof of Lemma 8 below we also make use of *history-dependent strategies*. A history-dependent strategy maps every play to an action and a delay such that the play is continued. Bouyer *et al.* [7] proved the following fundamental theorem, showing that history-dependent strategies are not needed for ϵ-optimal play, for any $\epsilon > 0$.

Theorem 6 (Bouyer *et al.* [7]). *For every 1PTG G, there exist value functions $v_k(x)$. Moreover, a player can get arbitrarily close to the values even when restricted to playing positional strategies.*

Solving 1PTGs with Resets. We reduce solving any 1PTG to solving a number of SPTGs. The first step towards this goal is to remove reset actions by extending the game.

Lemma 8. *Let G be an (n, m, r, d)-1PTG. Solving G can be reduced to solving $r + 1$ $(n, m, 0, d)$-1PTGs.*

The idea of the proof of Lemma 8 is that if a play, defined by some ϵ-optimal positional strategy profile for G, uses more than r reset actions, then the same configuration $(k, 0)$, for some state k, appears twice in the play. Since the strategies are positional the play must repeatedly cycle back to $(k, 0)$, and the play is, in fact, of infinite length. Thus, when playing G we may augment configurations by the number of times a reset action has been used, and once this

number reaches $r+1$ we may assume without loss of generality that the value is infinite. This defines a new PTG G' with states $S' = S \times \{0, \ldots, r\}$ and actions $A' = A \times \{0, \ldots, r\}$. An ϵ-optimal positional strategy profile for G' can then be viewed as an ϵ-optimal history-dependent strategy profile for G. Theorem 6 implies that the values obtained from G' are the same as the values for G. Due to the special structure of G' it can be solved as $r+1$ $(n, m, 0, d)$-1PTGs.

Solving 1PTGs without Resets. Let G be a 1PTG without reset actions, and let X be the set which consists of 0 and the endpoints of existence intervals of actions of G. Let the i'th largest element in X be M_i. Note that $M_1 = M$ and that $|X| \leq 2m+1$. Since there are no reset actions in G, it can be solved using the sweep-line technique described in Section 3.1. That is, we construct the optimal value functions starting from time M and moving to time 0. Note that the same actions are available throughout the open interval defined by two adjacent elements M_i and M_{i+1} of X. The situation within this interval is essentially identical to the situation in an SPTG, and this will be the key idea for how to solve G.

To be more precise we divide the interval from 0 to M into the points of X and the open intervals between the points of X. Starting from M and going to 0 we then solve a priced game for every point of X and an SPTG for every open interval. We use the previously computed value functions to define the games as we go along. Note that initially, at time M, it is not possible to wait, and we can get the correct values by solving a priced game.

Technically, we make the reduction from 1PTGs without resets to solving a number of SPTGs in three steps. First we reduce 1PTGs without resets to solving a number of 1PTGs where the existence interval of every action is either $(0, 1)$ or $[1, 1]$. A similar reduction was also used by Laroussinie, Markey, and Schnoebelen [13]. Note that it is always possible to scale time by scaling the rates appropriately. Next, we reduce specialized 1PTGs to 1PTGs with existence intervals $[0, 1]$ and $[1, 1]$, and finally we reduce these 1PTGs to SPTGs.

The proofs of the following lemma and theorems are in the full version [9]. Note that d is bounded by $2m+1$ and r is bounded by n.

Lemma 9. *Any (n, m, r, d)-1PTG G can be solved in time $O((r+1)d(n \log n + \min\{m, n^2\}))$ using at most $(r+1)d$ calls to an oracle that solves SPTGs with $n+1$ states and at most $m+n+1$ actions.*

Theorem 7. *Any (n, m, r, d)-1PTG G can be solved in time*

$$O((r+1)d(\min(m, n^2) + n \cdot \min\{12^n, \prod_{k \in S}(A_k + 1)\})).$$

Theorem 8. *The BCFL-ABM algorithm solves any 1PTG G using at most $m \cdot n^{O(1)} \min\{12^n, \prod_{k \in S}(A_k + 1)\})$ iterations.*

Theorem 9. *Any (n, m, r, d)-1PTG G, where all states have rate 1 and all actions have cost 0, can be solved in time $O((r+1)d(n \log n + \min(m, n^2)))$.*

Theorem 10. *Every priced timed automata G (i.e., all states are controlled by Player 1) that is a (n, m, r, d)-1PTG can be solved in time $O((r + 1)dn^2(\min(m, n^2) + n \log n))$.*

Acknowledgements. We would like to thank Kim Guldstrand Larsen for many helpful discussions and comments.

References

1. Alur, R., Bernadsky, M., Madhusudan, P.: Optimal reachability for weighted timed games. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 122–133. Springer, Heidelberg (2004)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
3. Alur, R., La Torre, S., Pappas, G.J.: Optimal paths in weighted timed automata. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 49–62. Springer, Heidelberg (2001)
4. Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J.M.T., Vaandrager, F.W.: Minimum-cost reachability for priced timed automata. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 147–161. Springer, Heidelberg (2001)
5. Bouyer, P., Brihaye, T., Markey, N.: Improved undecidability results on weighted timed automata. Inf. Process. Lett. 98, 188–194 (2006)
6. Bouyer, P., Cassez, F., Fleury, E., Larsen, K.G.: Optimal strategies in priced timed game automata. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 148–160. Springer, Heidelberg (2004)
7. Bouyer, P., Larsen, K.G., Markey, N., Rasmussen, J.I.: Almost optimal strategies in one clock priced timed games. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 345–356. Springer, Heidelberg (2006)
8. Brihaye, T., Bruyère, V., Raskin, J.-F.: On optimal timed strategies. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 49–64. Springer, Heidelberg (2005)
9. Hansen, T.D., Ibsen-Jensen, R., Miltersen, P.B.: A faster algorithm for solving one-clock priced timed games. CoRR, abs/1201.3498 (2012)
10. Howard, R.: Dynamic programming and Markov processes. MIT Press (1960)
11. Jurdziński, M., Trivedi, A.: Reachability-time games on timed automata. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 838–849. Springer, Heidelberg (2007)
12. Khachiyan, L., Boros, E., Borys, K., Elbassioni, K., Gurvich, V., Rudolf, G., Zhao, J.: On short paths interdiction problems: Total and node-wise limited interdiction. Theory of Computing Systems 43, 204–233 (2008)
13. Laroussinie, F., Markey, N., Schnoebelen, P.: Model checking timed automata with one or two clocks. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 387–401. Springer, Heidelberg (2004)
14. Rutkowski, M.: Two-player reachability-price games on single-clock timed automata. In: Proc. of 9th QAPL, pp. 31–46 (2011)
15. Shamos, M.I., Hoey, D.: Geometric intersection problems. In: Proc. of 17th FOCS, pp. 208–215 (1976)
16. Torre, S.L., Mukhopadhyay, S., Murano, A.: Optimal-reachability and control for acyclic weighted timed automata. In: Proc. of 2nd TCS, pp. 485–497 (2002)

Robust Controller Synthesis in Timed Automata

Ocan Sankur[1], Patricia Bouyer[1], Nicolas Markey[1], and Pierre-Alain Reynier[2]

[1] LSV, ENS Cachan & CNRS, France
[2] LIF, Aix-Marseille Université & CNRS, France

Abstract. We consider the fundamental problem of Büchi acceptance in timed automata in a robust setting. The problem is formalised in terms of controller synthesis: timed automata are equipped with a parametrised game-based semantics that models the possible perturbations of the decisions taken by the controller. We characterise timed automata that are robustly controllable for some parameter, with a simple graph theoretic condition, by showing the equivalence with the existence of an aperiodic lasso that satisfies the winning condition (aperiodicity was defined and used earlier in different contexts to characterise convergence phenomena in timed automata). We then show decidability and PSPACE-completeness of our problem.

1 Introduction

Timed automata [AD94] are a timed extension of finite-state automata, providing an automata-theoretic framework to design, model, verify and synthesise systems with timing constraints. However, the semantics of timed automata is an idealisation of real timed systems; it assumes, for instance, perfect clocks for arbitrarily precise time measures, and instantaneous actions. Thus, properties proven on timed automata may not hold in a real implementation, and similarly, a synthesised controller may not be realisable on a real hardware. This problem has been addressed in several works in the literature, where the goal is to define a convenient notion of *robustness*, so as to define a realistic semantics for timed automata, and also make sure that the verified (or synthesised) behaviour remains correct in presence of small perturbations.

In this work, we consider the fundamental problem of Büchi acceptance of a given timed automaton in a robust setting. Our goal is to distinguish timed automata where a Büchi condition can be satisfied even when the chosen delays are systematically perturbed by an adversary by a bounded parametrised amount. In fact, it has been observed that some timed automata require choosing time delays with infinite precision in order to realise some behaviours. Apart from well-known Zeno behaviours, [CHR02] shows such a convergence phenomenon where an infinite run requires increasing precision at each step. These unrealisable behaviours are discarded in such an adversarial robust setting. Thus, we

This work was partly supported by ANR projects ImpRo (ANR-2010-BLAN-0317) and ECSPER (ANR-2009-JCJC-0069), by ERC Starting grant EQualIS (308087), and by European project Cassting (FP7-ICT-601148).

P.R. D'Argenio and H. Melgratti (Eds.): CONCUR 2013, LNCS 8052, pp. 546–560, 2013.

formalize the problem in a game-theoretic setting. Our objective is to synthesise controllers that are robust while discarding unrealisable behaviours.

More precisely, to define robustness, we equip timed automata with the following game semantics between two players ([CHP11]): *Controller* with a given Büchi objective, and *Perturbator* with the complementary objective. The semantics is a turn-based game parametrised by $\delta > 0$. At each step, Controller chooses an edge, and $d > \delta$, such that the guard of the edge is satisfied after any delay $d' \in [d - \delta, d + \delta]$. Then, the edge is taken after a delay $d' \in [d - \delta, d + \delta]$ chosen by Perturbator. Timed games with parity conditions were studied in [CHP11] for a fixed known parameter $\delta > 0$, and for strictly positive delays with no lower bound. In fact, in this case, one can encode this semantics as a usual timed game, and rely on existing techniques for solving timed games. For an unknown parameter $\delta > 0$, the problem is more complicated, and was left as a challenging open problem in [CHP11].

Our main result is the following: we show that deciding the existence of $\delta > 0$, and of a strategy for Controller in the perturbation game so as to ensure infinite runs satisfying a given Büchi condition is PSPACE-complete, thus no harder than in the exact setting [AD94]. We characterise *robust* timed automata, *i.e.*, those in which Controller has a winning strategy, by showing that Controller can win precisely when the timed automaton has an accepting *aperiodic* lasso. Aperiodicity [Sta12] is a variant of *forgetfulness* introduced in [BA11] in a different context, to study the entropy of timed languages. Our characterisation confirms the suggestion of [BA11] that this notion could be significant in the study of robustness. Our results rely on the non-trivial combination of various techniques used for studying timed automata: forgetful and aperiodic cycles as considered in [BA11, Sta12], the metrics of [GHJ97], shrinking techniques [SBM11, BMS12] and reachability relations of [Pur00]. Last, our proof provides a symbolic representation of Controller's strategy which could be amenable to implementation.

A full version of the paper is available in [SBMR13].

Related Works. A similar game semantics was considered in [BMS12], but the winning objectives considered are only reachability. An important consequence is that convergence phenomena and unrealisable strategies are not an issue, since one essentially only considers finite paths. In this paper, we thus need new proof techniques to deal with convergence. In addition, the semantics considered in [BMS12] is less restrictive for Controller: he only needs to suggest delays after which the guard of the chosen edge is satisfied. Hence, the guard may not be satisfied after a perturbation. The emphasis in the resulting semantics is therefore on the newly appearing behaviours. Algorithmically, the semantics of [BMS12] gives rise to more complex problems: reachability is already EXPTIME-complete, whereas we are able to treat richer Büchi objectives in PSPACE in this paper. From a designer's perspective, we believe that both semantics are meaningful in different modelling assumptions. The present semantics is interesting if lower and upper bounds on events, *e.g* task execution times, appear naturally in the model, and need be respected strictly. On the other hand, in other applications, the semantics of [BMS12] allows to model

with equality constraints, and then synthesise controllers taking into account additional behaviour due to perturbations.

A related line of work is that of [Pur00, DDR05, DDMR08], which consists in modeling imprecisions by *enlarging* all clock constraints of the automaton by some parameter δ, that is, transforming each constraint of the form $x \in [a, b]$ into $x \in [a - \delta, b + \delta]$. The dual notion of *shrinking* was considered in [SBM11] in order to study whether any significant behaviour is lost when guards are shrunk. Both approaches are interested in model-checking, and the robustness condition is defined on the global behaviour of the enlarged or shrunk timed automaton. This does not capture the system's ability to adapt to perturbations that were observed earlier in a given run. In contrast, the game semantics endows Controller with a *strategy* against perturbations.

Among other robustness notions, [GHJ97] defines the *tube semantics* using a topology on timed automata runs. Our semantics is not related as we have a game semantics and a concrete parameter δ. However we use some results from [GHJ97] in our proofs. [Mar11] surveys different robust semantics for timed automata.

2 Timed Automata and Robust Safety

Given a finite set of clocks $\mathcal{C}$, we call *valuations* the elements of $\mathbb{R}_{\geq 0}^{\mathcal{C}}$. For a subset $R \subseteq \mathcal{C}$ and a valuation ν, $\nu[R \leftarrow 0]$ is the valuation defined by $\nu[R \leftarrow 0](x) = \nu(x)$ for $x \in \mathcal{C} \setminus R$ and $\nu[R \leftarrow 0](x) = 0$ for $x \in R$. Given $d \in \mathbb{R}_{\geq 0}$ and a valuation ν, the valuation $\nu + d$ is defined by $(\nu + d)(x) = \nu(x) + d$ for all $x \in \mathcal{C}$. We extend these operations to sets of valuations in the obvious way. We write $\mathbf{0}$ for the valuation that assigns 0 to every clock.

An atomic clock constraint is a formula of the form $k \preceq x \preceq' l$ or $k \preceq x - y \preceq' l$ where $x, y \in \mathcal{C}$, $k, l \in \mathbb{Z} \cup \{-\infty, \infty\}$ and $\preceq, \preceq' \in \{<, \leq\}$. A *guard* is a conjunction of atomic clock constraints. A valuation ν satisfies a guard g, denoted $\nu \models g$, if all constraints are satisfied when each $x \in \mathcal{C}$ is replaced with $\nu(x)$. We write $\Phi_{\mathcal{C}}$ for the set of guards built on $\mathcal{C}$.

A *timed automaton* $\mathcal{A}$ is a tuple $(\mathcal{L}, \mathcal{C}, \ell_0, E)$, where $\mathcal{L}$ is a finite set of locations, $\mathcal{C}$ is a finite set of clocks, $E \subseteq \mathcal{L} \times \Phi_{\mathcal{C}} \times 2^{\mathcal{C}} \times \mathcal{L}$ is a set of edges, and $\ell_0 \in \mathcal{L}$ is the initial location. An edge $e = (\ell, g, R, \ell')$ is also written as $\ell \xrightarrow{g,R} \ell'$.

The set of possible behaviours of a timed automaton can be described by the set of its runs, as follows. A *run* of $\mathcal{A}$ is a sequence $q_1 e_1 q_2 e_2 \dots$ where $q_i \in \mathcal{L} \times \mathbb{R}_{\geq 0}^{\mathcal{C}}$, and writing $q_i = (\ell, \nu)$, either $e_i \in \mathbb{R}_{>0}$, in which case $q_{i+1} = (\ell, \nu + e_i)$, or $e_i = (\ell, g, R, \ell') \in E$, in which case $\nu \models g$ and $q_{i+1} = (\ell', \nu[R \leftarrow 0])$. We denote by $\mathsf{state}_i(\rho)$ the i-th state of any run ρ, by $\mathsf{first}(\rho)$ its first state, and, if ρ is finite, $\mathsf{last}(\rho)$ denotes the last state of ρ.

In order to define perturbations, and to capture the reactivity of a controller to these, we define the following robust game semantics, defined in [CHP11] (see also [BMS12] for a variant). Intuitively, the robust semantics of a timed automaton is a two-player game parametrised by $\delta > 0$, where Player 1, also called *Controller* chooses a delay $d > \delta$ and an edge whose guard is satisfied

after any delay in the set $d + [-\delta, \delta]$. Then, Player 2, also called *Perturbator* chooses an actual delay $d' \in d + [-\delta, \delta]$ after which the edge is taken. Hence, Controller is required to always suggest delays that satisfy the guards whatever the perturbations are.

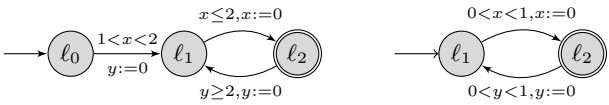
Formally, given a timed automaton $\mathcal{A} = (\mathcal{L}, \mathcal{C}, \ell_0, E)$ and $\delta > 0$, we define the *perturbation game* of $\mathcal{A}$ w.r.t. δ as a two-player turn-based game $\mathcal{G}_\delta(\mathcal{A})$ between players Controller and Perturbator. The state space of $\mathcal{G}_\delta(\mathcal{A})$ is partitioned into $V_C \cup V_P$ where $V_C = \mathcal{L} \times \mathbb{R}^{\mathcal{C}}_{\geq 0}$ belong to Controller, and $V_P = \mathcal{L} \times \mathbb{R}^{\mathcal{C}}_{\geq 0} \times \mathbb{R}_{\geq 0} \times E$ belong to Perturbator. The initial state is $(\ell_0, \mathbf{0}) \in V_C$. The transitions are defined as follows: from any state $(\ell, \nu) \in V_C$, there is a transition to $(\ell, \nu, d, e) \in V_P$ whenever $d > \delta$, $e = (\ell, g, R, \ell')$ is an edge such that $\nu + d + \varepsilon \models g$ for all $\varepsilon \in [-\delta, \delta]$. Then, from any such state $(\ell, \nu, d, e) \in V_P$, there is a transition to $(\ell', (\nu + d + \varepsilon)[R \leftarrow 0]) \in V_C$, for any $\varepsilon \in [-\delta, \delta]$. A pair of states of $V_C \cup V_P$ is said to be *consecutive* if there is a transition between them. A *play* of $\mathcal{G}_\delta(\mathcal{A})$ is a finite or infinite sequence $q_1 e_1 q_2 e_2 \ldots$ of states and transitions of $\mathcal{G}_\delta(\mathcal{A})$, with $q_1 = (\ell_0, \mathbf{0})$, where e_i is a transition from q_i to q_{i+1}. It is said to be *maximal* if it is infinite or cannot be extended. A *strategy* for Controller is a function that assigns to every non-maximal play ending in some $(\ell, \nu) \in V_C$, a pair (d, e) where $d > \delta$ and e is an edge such that there is a transition from (ℓ, ν) to (ℓ, ν, d, e). A strategy for Perturbator is a function that assigns, to every play ending in (ℓ, ν, d, e), a state (ℓ', ν') such that there is a transition from the former to the latter state. A play ρ is *compatible* with a strategy f if for every prefix ρ' of ρ ending in V_C, the next transition along ρ after ρ' is given by f. We define similarly compatibility for Perturbator's strategies. A play naturally gives rise to a unique run, where the states are in V_C, the delays are those chosen by Perturbator, and the edges are chosen by Controller.

Given $\delta > 0$, and a pair of strategies f, g, respectively for Controller and Perturbator we let $\mathsf{Outcome}^\delta_{\mathcal{A}}(f, g)$ denote the unique maximal run that is compatible with both strategies. We also define $\mathsf{Outcome}^\delta_{\mathcal{A}}(f, \cdot)$ (resp. $\mathsf{Outcome}^\delta_{\mathcal{A}}(\cdot, g)$) as the set of all maximal runs compatible with f (resp. with g). A *Büchi objective* is a subset of the locations of $\mathcal{A}$. Controller's strategy f is winning for a Büchi objective B, if all runs of $\mathsf{Outcome}^\delta_{\mathcal{A}}(f, \cdot)$ are infinite and visit infinitely often some location of B. The *parametrised robust controller synthesis problem* asks, given a timed automaton $\mathcal{A}$ and a Büchi objective B, whether there exists $\delta > 0$ such that Controller has a winning strategy in $\mathcal{G}_\delta(\mathcal{A})$ for the objective B. Note that these games are determined since for each $\delta > 0$, the semantics is a timed game.

Figure 1 shows examples of controllable and uncontrollable timed automata, in our sense. The main result of this paper is the following.

Theorem 1. *Parametrised robust controller synthesis is PSPACE-complete for Büchi objectives.*

The next section introduces several notions we need to state our main lemma (Lemma 3), which characterises timed automata that are robustly controllable, based on the nature of the lassos of the region automata.



(a) (b)

Fig. 1. On the left, a timed automaton from [Pur00] that is not robustly controllable for the Büchi objective $\{\ell_2\}$. In fact, Perturbator can enforce that the value of x be increased by δ at each arrival at ℓ_1, thus blocking the run eventually. On the right, the timed automaton (from [BA11]) is robustly controllable for the Büchi objective $\{\ell_2\}$. In fact, perturbations at a given transition do not affect the rest of the run; they are *forgotten*.

3 Regions, Orbit Graphs, Topology

Regions and Region Automata. We assume that the clocks are bounded above by a known constant in all timed automata we consider. Fix a timed automaton $\mathcal{A} = (\mathcal{L}, \mathcal{C}, \ell_0, \mathcal{E})$. We define *regions* as in [AD94], as subsets of $\mathbb{R}^{\mathcal{C}}_{\geq 0}$. Any region r is defined by fixing the integer parts of the clocks, and giving a partition $X_0, X_1, \ldots, X_m$ of the clocks, ordered according to their fractional values: for any $\nu \in r$, $0 = \mathsf{frac}(\nu(x_0)) < \mathsf{frac}(\nu(x_1)) < \ldots < \mathsf{frac}(\nu(x_m))$ for any $x_0 \in X_0, \ldots, x_m \in X_m$, and $\mathsf{frac}(\nu(x)) = \mathsf{frac}(\nu(y))$ for any $x, y \in X_i$. Here, $X_i \neq \emptyset$ for all $1 \leq i \leq m$ but X_0 might be empty. For any valuation ν, let $[\nu]$ denote the region to which ν belongs.

We define the *region automaton* $\mathcal{R}(\mathcal{A})$ as a finite automaton whose states are pairs (ℓ, r) where $\ell \in \mathcal{L}$ and r is a region. There is a transition $(\ell, r) \xrightarrow{\Delta} (\ell, s)$ if there exist $\nu \in r$, $\nu' \in s$ and $d > 0$ such that $\nu' = \nu + d$. There is a transition $(\ell, r) \xrightarrow{e} (\ell', s)$ where $e = (\ell, g, R, \ell')$ if $r \models g$ and $r[R \leftarrow 0] = s$. We write the *paths* of the region automaton as $\pi = q_1 e_1 q_2 e_2 \ldots q_n$ where each q_i is a state, and $e_i \in E \cup \{\Delta\}$, such that $q_i \xrightarrow{e_i} q_{i+1}$ for all $1 \leq i \leq n-1$. We also write $\mathsf{first}(\pi) = q_1$, $\mathsf{last}(\pi) = q_n$, $\mathsf{state}_i(\pi) = q_i$, and $\mathsf{trans}_i(\pi) = e_i$. The *length* of the path is n, and is denoted by $|\pi|$. We denote the subpath of π between states of indices i and j by $\pi_{i \ldots j}$. Given a run ρ of $\mathcal{A}$, its *projection on regions* is the path π in the region automaton s.t. $\mathsf{state}_i(\rho) \in \mathsf{state}_i(\pi)$ for all $1 \leq i \leq n$, and either $\mathsf{trans}_i(\rho) = \mathsf{trans}_i(\pi)$ or $\mathsf{trans}_i(\pi) = \Delta$ and $\mathsf{trans}_i(\rho) \in \mathbb{R}_{\geq 0}$. In this case, we write $\mathsf{first}(\rho) \xrightarrow{\pi} \mathsf{last}(\rho)$ (and say that ρ is *along* π). A *lasso* is a path $\pi_0 \pi_1$ where π_1 is a cycle, *i.e.* $\mathsf{first}(\pi_1) = \mathsf{last}(\pi_1)$. A *cycle* of $\mathcal{R}(\mathcal{A})$ is a *progress cycle* if it resets all clocks at least once [Pur00].

A region r is said to be *non-punctual* if it contains some $\nu \in r$ such that $\nu + [-\varepsilon, \varepsilon] \subseteq r$ for some $\varepsilon > 0$. It is said *punctual* otherwise. By extension, we say that (ℓ, r) is non-punctual if r is. A path $\pi = q_1 e_1 q_2 e_2 \ldots q_n$ is *non-punctual* if whenever $e_i = \Delta$, q_{i+1} is non-punctual.

Vertices and Orbit Graphs. A *vertex* of a region r is any point of $\bar{r} \cap \mathbb{N}^{\mathcal{C}}$, where $\bar{r}$ denotes the topological closure of r. For any region r, and any clock x, let us denote by $r_{x,0}$ the upper bound (by $-r_{0,x}$ the lower bound) on clock x

inside region r. Then, a vertex v of r is defined by the choice of an index $0 \leq i \leq m$ such that for all $x \in X_1, \ldots, X_i$, we have $v(x) = -r_{0,x}$ and for all $x \in X_{i+1}, \ldots, X_m$, we have $v(x) = r_{x,0}$. Hence, any region has at most $|\mathcal{C}| + 1$ vertices (See e.g. [DDMR08]). We denote by $\inf(r)$ (resp. $\sup(r)$) the vertex of r where all clocks are equal to their lower bounds (resp. upper bounds). Let $\mathcal{V}(r)$ denote the set of vertices of r. We also extend this definition to $\mathcal{V}((\ell, r)) = \mathcal{V}(r)$. Note the following easy properties of regions. Any region r has at most one vertex $v \in \mathcal{V}(r)$ such that both v and $v + 1$ belong to $\mathcal{V}(r)$. If these exist, then $v = \inf(r)$ and $v + 1 = \sup(r)$. Moreover, $\sup(r) = \inf(r) + 1$ if, and only if r is non-punctual. It has been shown that all valuations in r are convex combinations of $\mathcal{V}(r)$ [Pur00].

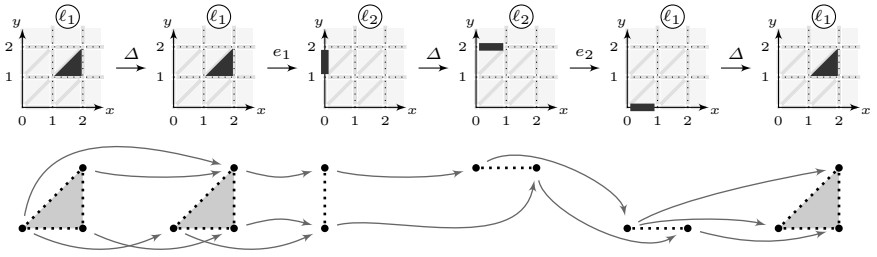


Fig. 2. The orbit graph of a (cyclic) path in the region automaton of the automaton of Fig. 1(a)

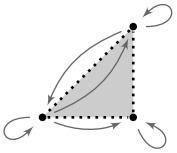


Fig. 3. The folded orbit graph of the (non-forgetful) cycle of Fig. 2

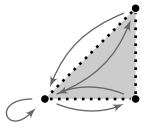


Fig. 4. The folded orbit graph of a forgetful cycle

With any path π of the region automaton, we associate a $|\pi|$-partite labelled graph $\gamma(\pi)$ called the *orbit graph of* π [Pur00]. Intuitively, the orbit graph of a path gives the reachability relation between the vertices of the regions visited along the path. Formally, for a transition $\tau = q_1 e_1 q_2$, its orbit graph $\gamma(\tau) = (V_1 \cup V_2, f_G, E)$ is a bipartite graph where $V_1 = \{(1, v)\}_{v \in \mathcal{V}(q_1)}$, and $V_2 = \{(2, v)\}_{v \in \mathcal{V}(q_2)}$. For any $\big((1, u), (2, v)\big) \in V_1 \times V_2$, we have an edge $((1, u), (2, v)) \in E$, if, and only if $u \xrightarrow{\overline{e_1}} v$, where $\overline{e_1} = \Delta$ if $e_1 = \Delta$, and otherwise $\overline{e_1}$ is obtained by replacing the guard by its closed counterpart. Note that each vertex has at least one successor through $\overline{e_1}$ [AD94]. The labelling function f_G maps each i to q_i; we also extend to nodes of G by $f_G((i, v)) = f_G(i)$. In order to extend $\gamma(\cdot)$ to paths, we use a composition operator $\oplus$ between orbit graphs, defined as follows. If $G = (V_1 \cup \ldots \cup V_n, f_G, E)$ and $G' = (V'_1 \cup \ldots \cup V'_m, f_{G'}, E')$

denote two orbit graphs, then $G \oplus G'$ is defined if, and only if, $f_G(n) = f_{G'}(1)$, that is, when the path defining the former graph ends in the first state of the path defining the latter graph. In this case, the graph $G'' = G \oplus G' = (V_1'' \cup \ldots V_{n+m-1}'', f_{G''}, E'')$ is defined by taking the disjoint union of G and G', merging each node (n, v) of V_n with the node $(1, v)$ of V_1', and renaming any node $(i, v) \in V_i'$ by $(i + n - 1, v)$, so that we get a $(n + m - 1)$-partite graph. Formally, we let $V_i = V_i''$ for all $1 \leq i \leq n$, and the subgraph of G'' induced on these nodes is equal to G. For any $n + 1 \leq i \leq n + m - 1$, we have $V_i'' = \{(i, v)\}_{(i-n+1,v)\in V_{i-n+1}'}$, and there is an edge $\big((i, v), (i + 1, w)\big) \in E''$ if, and only if, $\big((i - n + 1, v), (i - n, w)\big) \in E'$. Now, we extend $\gamma(\cdot)$ to paths by induction, as follows. Consider any path $\pi = q_1 e_1 \ldots q_{n-1} e_{n-1} q_n$, and let $G = (V_1 \cup \ldots \cup V_{n-1}, f_G, E)$ be the $(n - 1)$-partite graph $\gamma(q_1 e_1 \ldots q_{n-1})$, given by induction. Let $G' = (U \cup U', f_{G'}, E')$ denote the bipartite graph of $q_{n-1} e_{n-1} q_n$. Then, we let $\gamma(\pi) = G \oplus G'$. For any node (i, v) of $\gamma(\pi)$, let $\mathsf{Succ}((i, v))$ denote the set of nodes $(i + 1, w)$ with $\big((i, v), (i + 1, w)\big)$ is an edge. We also extend $\mathsf{Succ}(\cdot)$ to sets of nodes. Fig. 2 displays a path in the region automaton of the automaton depicted on Fig. 1(a) together with its orbit graph. Note that delays of duration zero are allowed when defining orbit graphs.

We define the *folded orbit graph* $\Gamma(\pi)$ for any path π that is not a cycle, as a bipartite graph on node set $\{1\} \times \mathcal{V}(\mathsf{first}(\pi)) \cup \{2\} \times \mathcal{V}(\mathsf{last}(\pi))$. There is an edge $\big((1, v), (2, w)\big)$ in $\Gamma(\pi)$ if, and only if there is a path from $(1, v)$ to (n, w) in $\gamma(\pi)$, where $n = |\pi|$. Nodes are labelled by the regions they belong to. For any cycle π, we define $\Gamma(\pi)$ similarly on the node set $\mathcal{V}(\mathsf{first}(\pi))$. Thus $\Gamma(\pi)$ may contain cycles; an example is given in Fig. 3. We extend the operator $\oplus$ to folded orbit graphs. A strongly connected component (SCC) of a graph is *initial* if it is not reachable from any other SCC.

A *forgetful cycle* of $\mathcal{R}(\mathcal{A})$ is a cycle whose folded orbit graph is strongly connected. A cycle π is *aperiodic* if for all $k \geq 1$, π^k is forgetful. A lasso is said to be aperiodic if its cycle is. Note that there exist forgetful cycles that are not aperiodic [Sta12].

An example of a non-forgetful cycle is given in Fig. 3. The timed automaton of Fig. 1(b) contains a forgetful cycle, shown on Fig. 4.

Some Linear Algebra. For any set of vectors, we denote by $\mathsf{Span}(\mathcal{B})$ the linear span of $\mathcal{B}$, *i.e.* the set of linear combinations of $\mathcal{B}$. In the proofs, we will often use the vertices of a region to define a basis of a vector space that contains the region.

Lemma 2. *Let r be any region, and let $v_0 = \inf(r)$. The set of vectors $\mathcal{B}_{v_0} = \{v - \inf(r)\}_{v\in\mathcal{V}(r)\setminus\{v_0\}}$ is linearly independent. Moreover, the affine space $v_0 + \mathsf{Span}(\mathcal{B}_{v_0})$ contains r.*

Let the dimension of a subset $r \subseteq \mathbb{R}^{\mathcal{C}}$ be the least d such that a affine subspace of $\mathbb{R}^{\mathcal{C}}$ of dimension d contains r. It follows immediately from Lemma 2 that in any region where the partition of the clocks according to their fractional values is written as $X_0, X_1, \ldots, X_m$, has dimension m since it has $m + 1$ vertices.

We will consider the usual d_∞ metric on $\mathbb{R}^{\mathcal{C}}$, defined as $d_\infty(\nu, \nu') = \max_{x\in\mathcal{C}} |\nu(x) - \nu'(x)|$. We denote open balls in this metric by $\mathsf{Ball}_{d_\infty}(\nu, \varepsilon)$.

4 Main Lemma and Algorithm

Our main result is based on the following lemma, which gives a characterization of robust timed automata using aperiodic lassos of region automata.

Lemma 3 (Main Lemma). *For any timed automaton $\mathcal{A}$ and Büchi objective B, there exists $\delta > 0$ such that Controller has a winning strategy in $\mathcal{G}_\delta(\mathcal{A})$ for objective B, if, and only if $\mathcal{R}(\mathcal{A})$ has a reachable aperiodic non-punctual B-winning lasso.*

The algorithm for deciding robust Büchi acceptance follows from Lemma 3. It consists in looking for aperiodic non-punctual B-winning lassos in $\mathcal{R}(\mathcal{A})$. These lassos need not be simple, but the following lemma bounds their lengths.

Lemma 4. *Let B be a Büchi objective in a timed automaton $\mathcal{A}$, and π be an aperiodic non-punctual B-winning cycle of $\mathcal{R}(\mathcal{A})$. Then, there exists a cycle π' with the same properties, with length at most $N = 2^{(|\mathcal{C}|+1)^2+1} \times |\mathcal{R}(\mathcal{A})|$.*

The polynomial-space algorithm then consists in guessing an accepting lasso of exponential size in $\mathcal{R}(\mathcal{A})$ on-the-fly, and checking whether its folded orbit graph is aperiodic. The folded orbit graph can also be computed on-the-fly, and aperiodicity can be checked in PSPACE [Sta12]. See Appendix for more details.

The two directions of the main lemma are proved using different techniques; they are presented respectively in Sections 5 and 6. Our results also establish that winning strategies can be represented by regions with a given granularity, depending on δ. An algorithm is described at the end of Section 6 to actually compute the bound δ and a description of the winning strategy for Controller.

5 No Aperiodic Lassos Implies No Robustness

In this section, we prove that Controller loses if there is no aperiodic winning lassos. The idea is that if no accepting lasso of $\mathcal{R}(\mathcal{A})$ is aperiodic, then, as we show, the projection of any play to $\mathcal{R}(\mathcal{A})$ eventually enters and stays in a non-forgetful cycle. Then, we choose an appropriate *Lyapunov function* $L_I(\cdot)$ defined on valuations and taking nonnegative values, and describe a strategy for Perturbator such that the value of $L_I(\cdot)$ is decreased by at least ε at each iteration of the cycle. Hence, Controller cannot cycle infinitely on such cycles: either it reaches a deadlock, or it cycles on non-accepting lassos. In the rest of this section, we describe Perturbator's strategy, study its outcomes, choose a function $L_I(\cdot)$, and prove the first direction of the main lemma.

Our proof is based on several results. First, we consider a result from Puri [Pur00](Lemma 5) on reachability relations between valuations in timed automata, and establish non-trivial properties on it valid along non-punctual paths. We then study the folded orbit graphs of non-punctual progress cycles, and use original proof techniques (*e.g.* using the dimension of sets) to understand the form of these graphs. This allows us to consider the Lyapunov functions of [BA11] in this context, and prove our results as described above.

5.1 Reachability Relations

We already noted that any valuation ν can be written as the convex combination of the vertices of its region, i.e. $\nu = \sum_{v \in \mathcal{V}([\nu])} \lambda_v v$ for some unique coefficients $\lambda_v \geq 0$ with $\sum_v \lambda_v = 1$. When the region is clear from context, we will simply write $\nu = \boldsymbol{\lambda v}$. Given a path π and a vertex v of the region of $\mathsf{first}(\pi)$, let us denote by $R_{\Gamma(\pi)}(v)$ the set of nodes $w \in \mathcal{V}(\mathsf{last}(\pi))$ such that $(v, w) \in E(\Gamma(\pi))$. Thus, this is the "image" of v by the path π. Puri showed in [Pur00] that the reachability along paths can be characterized using orbit graphs.

Lemma 5 ([Pur00]). *Let π be a path from region r to s. Consider any $\nu \in r$ with $\nu = \sum_{v \in \mathcal{V}(r)} \lambda_v v$ for some coefficients $\lambda_v \geq 0$ and $\sum \lambda_v = 1$. If $\nu \xrightarrow{\pi} \nu'$, then for each $v \in \mathcal{V}(r)$, there exists a probability distribution $\{p_{v,w}^{\nu,\nu'}\}_{w \in R_{\Gamma(\pi)}(v)}$ over $R_{\Gamma(\pi)}(v)$ such that $\nu' = \sum_{v \in \mathcal{V}(r)} \lambda_v \sum_{w \in R_{\Gamma(\pi)}(v)} p_{v,w}^{\nu,\nu'} w$. Conversely, if there exist probability distributions $p_{v,w}^{\nu,\nu'}$ satisfying above equalities, then $\nu \xrightarrow{\bar{\pi}} \nu'$.*

Intuitively, the lemma shows that any successor of a point $\nu = \sum_i \lambda_i v_i$ can be obtained by distributing the weight λ_i of each vertex v_i to its successors following a probability distribution.

Example 1. The automaton of Fig. 1(a) contains a cycle on the region $r = [\![1 < x, y < 2 \wedge 0 < x - y < 1]\!]$. The vertices of r are $v_1 = (1, 1), v_2 = (2, 1), v_3 = (2, 2)$. Consider a point $\nu = \frac{1}{3}v_1 + \frac{1}{3}v_2 + \frac{1}{3}v_3$. Then, Lemma 5 says that $\nu' = \sum_{1 \leq i \leq 3} \lambda_i v_i$ is reachable from ν along the cycle, where $\lambda_1 = \frac{1}{3}0.5 + \frac{1}{3}0.4 = \frac{9}{30}$, $\lambda_2 = \frac{1}{3}1 + \frac{1}{3}0.3 = \frac{13}{30}$, and $\lambda_3 = \frac{1}{3}0.6 + \frac{1}{3}0.2 = \frac{4}{15}$. Here, vertex set $\{v_1, v_3\}$ is an initial SCC I. One can check that L_I is indeed decreasing: $\frac{1}{3} + \frac{1}{3} \geq \frac{9}{30} + \frac{4}{15}$.

For any region r, and any subset $I \subseteq \mathcal{V}(r)$, we define the function $L_I : \bar{r} \to \mathbb{R}_{\geq 0}$ as, $L_I(\nu) = \sum_{v \in I} \lambda_v$, where $\nu = \boldsymbol{\lambda v}$. It is shown in [BA11] that given any cycle π, if I is chosen as the initial strongly connected component of $\Gamma(\pi)$, then for any run $\nu \xrightarrow{\pi} \nu'$, $L_I(\nu') \leq L_I(\nu)$. We will abusively use $L_I(\cdot)$ for a subset I of nodes of $\gamma(\pi)$ or $\Gamma(\pi)$, that correspond to a same region. Notice that $0 \leq L_I(\cdot) \leq 1$.

5.2 A Global Strategy for Perturbator

Let us call a valuation v *ε-far* if $v + [-\varepsilon, \varepsilon] \subseteq [v]$. A run is ε-far if all delays end in ε-far valuations. We show that Perturbator has a strategy ensuring ε-far runs.

Lemma 6. *Given any $\delta > 0$, and any timed automaton with C clocks, there exists a strategy $\sigma^P_{\delta,+}$ (resp. $\sigma^P_{\delta,-}$) for Perturbator that always perturbs by a positive (resp. negative) amount, and whose all outcomes are $\frac{\delta}{2(C+1)}$-far, and all delays are at least $\frac{\delta}{2(C+1)}$.*

Proof. After any delay $\nu \xrightarrow{d} \nu'$, $d \geq \delta$, chosen by Controller, consider the regions spanned by the set $\nu' + [0, \delta]$. It is easy to see that this set intersects at most

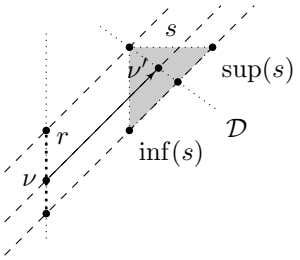
$|\mathcal{C}|+1$ different regions (See also [BMS12, Lemma 6]), all of which must satisfy the guard by definition of the game. So some region r satisfies $\nu' + [\alpha, \beta] \subseteq r$, for some $0 \leq \alpha < \beta \leq \delta$ with $\beta - \alpha \geq \frac{\delta}{|\mathcal{C}|+1}$. The strategy $\sigma^{\mathsf{P}}_{\delta,+}$ consists in choosing the perturbation as $\frac{1}{2}(\beta-\alpha)$. This guarantees time progress (of at least $\frac{\delta}{2(|\mathcal{C}|+1)}$). Moreover, the resulting valuation is always ε-far in its region, where $\varepsilon = \frac{\delta}{2(|\mathcal{C}|+1)}$. Observe also that all perturbations prescribed by this strategy are positive. The strategy $\sigma^{\mathsf{P}}_{\delta,-}$ is constructed similarly by considering the valuations $\nu' + [-\delta, 0]$.

It turns out that in order to win, Perturbator only needs to ensure ε-far runs, hence either of the strategies defined above is sufficient to win. In the rest, let us fix a strategy $\sigma^{\mathsf{P}}_{\delta}$ as $\sigma^{\mathsf{P}}_{\delta,+}$ or $\sigma^{\mathsf{P}}_{\delta,-}$. In order to prove that ε-far runs are winning for Perturbator, we study the properties of the runs $\mathsf{Outcome}^{\delta}_{\mathcal{A}}(\cdot, \sigma^{\mathsf{P}}_{\delta})$. We prove Propositions 7 and 8 which are a key element of the proof.

Using the ε-far property of the runs, the following proposition derives a bound on the convex combination coefficients of all visited valuations.

Proposition 7. *Let $\rho \in \mathsf{Outcome}_{\mathcal{A}}(\cdot, \sigma^{P}_{\delta})$. For any $i \geq 1$, if we write $\mathsf{state}_i(\rho) = \boldsymbol{\lambda v}$, then $\lambda_v \geq \varepsilon$ for all vertices $v \in \mathcal{V}([\mathsf{state}_i(\rho)])$.*

Intuitively, a lower bound on the convex coefficients means that the valuation is not close to the borders of the region. We sketch the proof which is by induction. The property is true initially since the region **0** has a single vertex. For the induction case, let us mention the easy case of resets. Clock resets map each vertex to a single vertex, so each vertex has a single successor in the orbit graph. Then, it follows from Lemma 5 that the coefficient of each vertex in the target region is at least as large as its predecessor in the source region. For the case of time delays, one needs to consider the geometry of regions. The figure on the right shows the intuition in two dimensions. Given an ε-far delay from ν to ν', with $[\nu] = r$, and $[\nu'] = s$, one shows that the coefficients of $\inf(s)$ and $\sup(s)$ cannot be too small; otherwise the line $\mathcal{D}$ that connects ν' to the third vertex would be close to vertical or to horizontal, requiring ν' to be close to a border of s.



We need another property of similar spirit stating that all edges of the folded orbit graph receive a probability of at least $\min(\frac{1}{2}, \frac{\varepsilon}{2})$ along ε-far delays, according to the interpretation of Lemma 5. The proof is rather involved, and establishes that there is some degree of freedom in the choice of the probabilities of Lemma 5.

Proposition 8. *Let $\nu = \boldsymbol{\lambda v}$ and $\nu' = \boldsymbol{\lambda' v'}$ denote two valuations satisfying $\boldsymbol{\lambda}, \boldsymbol{\lambda'} \geq \varepsilon$, and s.t. $(\ell, \nu) \xrightarrow{\pi} (\ell, \nu')$ is an ε-far delay of duration at least ε. Then, for each $v \in \mathcal{V}([\nu])$, there exists a probability distribution $\{p_{v,w}\}_{w \in R_{\Gamma(\pi)}(v)}$ over $R_{\Gamma(\pi)}(v)$ s.t. $\nu' = \sum_{v \in \mathcal{V}(r)} \lambda_v \sum_{w \in R_{\Gamma(\pi)}(v)} p_{v,w} w$, and $p_{v,w} \geq \min(\frac{1}{2}, \frac{\varepsilon}{2})$.*

5.3 Decreasing Lyapunov Function

In the previous subsection, we established lower bounds on the convex coefficients of the visited valuations, and the probabilities of Lemma 5. We use this property to find a Lyapunov function that strictly decreases at each iteration of a cycle.

In this subsection, we concentrate on progress cycles. In fact, in the proof of Lemma 3, we will show that if Controller is able to win against $\sigma_\delta^{\mathsf{P}}$, then some cycle of $\mathcal{R}(\mathcal{A})$ must be repeated infinitely often. But since $\sigma_\delta^{\mathsf{P}}$ always ensures a time progress of $\frac{\delta}{2(|\mathcal{C}|+1)}$ (see Lemma 6), and because all clocks are bounded, this is only possible in a progress cycle.

The following lemma shows that along non-punctual progress cycles, runs can reach any valuation in a ball around the target state. This gives the dimension of the set of valuations reachable along a progress cycle starting from a given valuation. The proof is somewhat similar to [DDMR08, Lemma 29].

Lemma 9. *Let π be a non-punctual progress cycle, and $(\ell,\nu) \xrightarrow{\pi} (\ell,\nu')$ a run along π. Then, there exists $\varepsilon > 0$ such that there exists a run from (ℓ,ν), along π, to any point in $\{\ell\} \times (\mathsf{Ball}_{d_\infty}(\nu',\varepsilon) \cap [\nu'])$.*

We now prove that the folded orbit graphs of non-punctual progress cycles are always connected. If the cycle is non-forgetful, there are at least two connected SCCs (Corollary 11). The lemma is proved by contradiction: we show that if the graph has disjoint components, then the set of states reachable from a given state cannot have full dimension, which contradicts Lemma 9.

Lemma 10. *The folded orbit graph of a non-punctual progress cycle is connected.*

Corollary 11. *The folded orbit graph of a non-punctual non-forgetful progress cycle π contains at least two strongly connected components that are connected. We associate with each π an initial SCC of $\Gamma(\pi)$, which we denote by $I(\pi)$.*

Hence, for any non-punctual non-forgetful cycle π, we consider the function $L_{I(\pi)}$. The following lemma shows a key property for the proof: $L_{I(\pi)}$ decreases by a fixed amount at each iteration of such a cycle under Perturbator's strategy $\sigma_\delta^{\mathsf{P}}$.

Lemma 12. *Let $\omega \in \mathsf{Outcome}_{\mathcal{A}}^{\delta}(\cdot,\sigma_\delta^P)$, and ρ be a finite prefix of ω such that π, the projection of ρ to regions, is a cycle. If π is a non-forgetful progress cycle, then, writing $\mathsf{first}(\rho) = \boldsymbol{\lambda v}$ and $\mathsf{last}(\rho) = \boldsymbol{\lambda' v'}$, we have, $\sum_{i\in I(\pi)} \lambda_i' \leq \sum_{i\in I(\pi)} \lambda_i - \varepsilon^2/2$.*

The proof shows that any such run has a transition in which some edge of the folded orbit graph has a successor to a node that is not coreachable from $I(\pi)$. The existence of such an edge is proved using Corollary 11. By Propositions 7 and 8, we know that the convex combination coefficient associated to the node is at least ε, and the probability associated to the edge leaving it is at least $\varepsilon/2$. One then shows that at least $\varepsilon^2/2$ is lost from L_π at each iteration.

The previous lemma already gives an insight into the proof, since it follows that no non-forgetful cycle can be repeated infinitely under strategy $\sigma_\delta^{\mathsf{P}}$. However, one also needs to show that switching between different cycles cannot help

Controller win. Thus, the last tool we need for the proof is the following factorization theorem, which allows factoring paths to cycles with the same folded orbit graphs.

Lemma 13. *Let π be a path of $\mathcal{R}(\mathcal{A})$ written as $\pi = \pi_0\pi_1\pi_2\dots\pi_n$ where each π_i is a cycle that starts in the same state, for $i \geq 1$. Then, one can write $\pi = \pi'_0\pi'_1\pi'_2\dots\pi'_{m+1}$ such that $m \geq \sqrt{n/r-2} - 1$, where $r = 2^{(|\mathcal{C}|+1)^2}|\mathcal{R}(\mathcal{A})|$, and for some indices $0 = \alpha_0 < \alpha_1 < \dots$, we have $\pi'_i = \pi_{\alpha_i} \cdot \dots \cdot \pi_{\alpha_{i+1}-1}$ for each $i \geq 0$, and $\Gamma(\pi'_1) = \Gamma(\pi'_i)$ for all $1 \leq i \leq m$.*

Proof: No winning aperiodic lassos implies no robust safety. To get a contradiction, fix any winning strategy σ for Controller and let ρ be the infinite run $\mathsf{Outcome}^{\delta}_{\mathcal{A}}(\sigma, \sigma^{\mathsf{P}}_{\delta})$, and π its projection on regions. By definition of $\sigma^{\mathsf{P}}_{\delta}$, π is a non-punctual path. Let us write $\pi = \pi_0\pi_1\dots\pi_n\dots$ such that all π_i, $i \geq 1$, are accepting cycles from a same state. Let $r = 2^{(|\mathcal{C}|+1)^2} \times |\mathcal{R}(\mathcal{A})|$, $n = \lceil 2/\varepsilon^2 \rceil + 1$ and N large enough so that $\lceil \sqrt{N/r-2} \rceil - 1 \geq n2^{(|\mathcal{C}|+1)^2}$. We extract π' the prefix of π with N factors, and apply Lemma 13 which yields $\pi' = \pi'_0\pi'_1\dots\pi'_{n'}\pi'_{n'+1}$, with $n' = n2^{(|\mathcal{C}|+1)^2}$, where $\Gamma(\pi'_1) = \dots = \Gamma(\pi'_{n'})$, and π'_i are obtained by concatenating one or several consecutive π_i. By hypothesis, some power k of π'_i is non-forgetful, with $k \leq 2^{(|\mathcal{C}|+1)^2}$ since this is the number of folded orbit graphs for a fixed labelling function. But since the folded orbit graphs are the same for all π'_i, this power is the same for all factors, and $\Gamma(\pi'^k_i) = \Gamma(\pi'_i\pi'_{i+1}\dots\pi'_{i+k-1})$. Hence, we can factorize π' again into $\pi' = \pi'_0\pi''_1\pi''_2\dots\pi''_n\pi''_{n+1}$, where $\Gamma(\pi''_1) = \dots = \Gamma(\pi''_n)$ and all are non-forgetful; while π'_0 and π''_{n+1} are arbitrary non-punctual paths. Moreover, π''_i, for $1 \leq i \leq n$, must be progress cycles too. In fact, otherwise there is some clock $x \in \mathcal{C}$ that is never reset along π'. But because $\sigma^{\mathsf{P}}_{\delta}$ ensures a time progress of ε at each delay, this means that π' contains delays of duration at least $n\varepsilon^2/2 > 1$, so we cannot have $\mathsf{first}(\pi'_1) = \mathsf{last}(\pi'_n)$ since the integer part of the clock x changes, and so does the region since all clocks are assumed to be bounded. Now, we have $I(\pi''_i) = I(\pi''_j)$ for any $1 \leq i, j \leq n$, so the functions $L_{I(\pi''_i)}$ are the same for all $1 \leq i \leq n$. If we write ρ_i the state reached in ρ following $\pi'_0\pi''_1\dots\pi''_i$, then we get, by Lemma 12, $L_{I(\pi''_1)}(\rho_n) \leq L_{I(\pi''_1)}(\rho_0) - n\varepsilon^2/2$. This is a contradiction since $0 \leq L_{I(\pi'_1)} \leq 1$ and $n\varepsilon^2/2 > 1$.

6 Aperiodic Lassos Implies Robustness

We now prove that if $\mathcal{R}(\mathcal{A})$ contains an aperiodic non-punctual B-winning lasso, then there exists $\delta > 0$ and a strategy for Controller in $\mathcal{G}_{\delta}(\mathcal{A})$ ensuring robust safety. The idea of the proof is to observe that aperiodic cycles do not constrain runs in the way non-forgetful ones do, and show that this is still the case in the perturbation game semantics. More precisely, along an aperiodic lasso, Controller is able to always come back in a set at the middle of the starting region.

6.1 Zones and Shrunk Zones

A *zone* is a subset of $\mathbb{R}^{\mathcal{C}}_{\geq 0}$ defined by a guard. A *difference-bound matrix (DBM)* is a $|\mathcal{C}_0| \times |\mathcal{C}_0|$-matrix over $(\mathbb{R} \times \{<, \leq\}) \cup \{(\infty, <)\}$. We adopt the following

notation: for any DBM M, we write $M = (\mathsf{M}, \prec^M)$, where M is the matrix made of the first components, with elements in $\mathbb{R} \cup \{\infty\}$, while $\prec^M$ is the matrix of the second components, with elements in $\{<, \leq\}$. A DBM M naturally represents a zone (which we abusively write M as well), defined as the set of valuations v such that, for all $x, y \in \mathcal{C}_0$, it holds $v(x) - v(y) \prec^M_{x,y} \mathsf{M}_{x,y}$ (where $v(0) = 0$). Standard operations used to explore the state space of timed automata have been defined on DBMs: intersection is written $M \cap N$, $\mathsf{Pre}\,(M)$ is the set of time predecessors of M, $\mathsf{Unreset}_R(M)$ is the set of valuations that end in M when the clocks in R are reset. We also consider $\mathsf{Pre}_{>\delta}(M)$, the set of time predecessors with a delay of more than δ. DBMs were introduced in [BM83, Dil90] for analyzing timed automata; we refer to [BY04] for details.

A parametrised extension, namely *shrunk DBMs* were introduced in [SBM11] in order to study the parametrised state space of timed automata. Intuitively, our goal is to express *shrinkings* of guards, *e.g.* sets of states satisfying constraints of the form $g = 1 + \delta < x < 2 - \delta \wedge 2\delta < y$, where δ is a parameter to be chosen. Formally, a shrunk DBM is a pair (M, P), where M is a DBM, and P is a nonnegative integer matrix called a *shrinking matrix* (SM). This pair represents the set of valuations defined by the DBM $M - \delta P$, for any given $\delta > 0$. Considering the example g, M is the guard g obtained by setting $\delta = 0$, and P is made of the integer multipliers of δ.

We adopt the following notation: when we write a statement involving a shrunk DBM (M, P), we mean that for some $\delta_0 > 0$, the statement holds for $(M - \delta P)$ for all $\delta \in [0, \delta_0]$. For instance, $(M, P) = \mathsf{Pre}_{>\delta}((N, Q))$ means that $M - \delta P = \mathsf{Pre}_{>\delta}(N - \delta Q)$ for all small enough $\delta > 0$. Additional operations are defined for shrunk DBMs: for any (M, P), we define $\mathsf{shrink}_{[-\delta,\delta]}((M, P))$ as the set of valuations ν such that $\nu + [-\delta, \delta] \subseteq M - \delta P$, for small enough $\delta > 0$.

Shrunk DBMs are closed under standard operations on zones:

Lemma 14 ([SBM11, BMS12]). *Let $M = f(N_1, \ldots, N_k)$ be an equation between normalized DBMs $M, N_1, \ldots, N_k$, using the operators $\mathsf{Pre}_{>\delta}$, $\mathsf{Unreset}_R$, $\cap$, and $\mathsf{shrink}_{[-\delta,\delta]}$, and let $P_1, \ldots, P_k$ be SMs. Then, there exists a shrunk DBM (M', Q) with $\mathsf{M}' = \mathsf{M}$, $M \subseteq M'$ and $(M', Q) = f\big((N_1, P_1), \ldots, (N_k, P_k)\big)$. The shrunk DBM (M', Q) and an upper bound on δ can be computed in poly-time.*

6.2 Controllable Predecessors

Consider an edge $e = (\ell, g, R, \ell')$. For any set $Z \subseteq \mathbb{R}^{\mathcal{C}}_{\geq 0}$, we define the *controllable predecessors of Z* as follows: $\mathsf{CPre}^{\delta}_e(Z) = \mathsf{Pre}_{>\delta}(\mathsf{shrink}_{[-\delta,\delta]}(g \cap \mathsf{Unreset}_R(Z)))$. Intuitively, $\mathsf{CPre}^{\delta}_e(Z)$ is the set of valuations from which Controller can ensure reaching Z in one step, following the edge e. In fact, it can delay in $\mathsf{shrink}_{[-\delta,\delta]}(g \cap \mathsf{Unreset}_R(Z))$ with a delay of more than δ, where under any perturbation in $[-\delta, \delta]$, the valuation satisfies the guard, and it ends, after reset, in Z. We extend this operator to paths as expected. Note that CPre^0_e is the usual predecessor operator without perturbation: If $N = \mathsf{CPre}^0_\pi(M)$ for some sets N, M and path π, then, N is the set of all valuations that can reach some valuation in M following π.

It immediately follows from Lemma 14 that the controllable predecessors of shrunk DBMs are shrunk DBMs, which are computable.

Corollary 15. *Let $e = (\ell, g, R, \ell')$ be an edge. Let M and N be non-empty DBMs such that $N = \mathsf{CPre}_e^0(M)$. Then, for any SM P, there exists an SM Q such that $(N', Q) = \mathsf{CPre}_e^\delta((M, P))$ for some $N \subseteq N'$ and $\mathsf{N} = \mathsf{N}'$.*

The set (N', Q) given by the previous lemma can be empty in general. However, as the following lemma shows, it turns out that in the case of non-punctual paths, controllable predecessors of open sets are non-empty, for small enough $\delta > 0$.

Lemma 16. *Let π be a non-punctual path from region r to s. Let $s' \subseteq s$ such that there exists $\nu' \in s'$ and $\varepsilon > 0$ with $\mathsf{Ball}_{d_\infty}(\nu', \varepsilon) \cap s \subseteq s'$. Then, $\mathsf{CPre}_\pi^\delta(s')$ is non-empty for small enough $\delta > 0$.*

6.3 Winning under Perturbations

Let $\pi_0\pi$ denote a non-punctual aperiodic lasso, where π is the cycle; $\Gamma(\pi^n)$ is strongly connected for $n \geq 1$. We prove that the graph of some power is complete:

Lemma 17. *Let π be an aperiodic cycle. Then, there exists $n \leq |\mathcal{C}_0| \cdot |\mathcal{C}_0|!$ such that $\Gamma(\pi^n)$ is a complete graph.*

Let us assume, by the previous lemma, that $\Gamma(\pi)$ is a complete graph (one can consider the lasso $\pi_0\pi^n$ for an appropriate n). Let s be a region with smaller granularity inside r, obtained so that the Hausdorff distance between r and s is positive. In this case, s can be chosen so that it can be expressed by a DBM (with rational components). The construction is defined in the following lemma.

Lemma 18. *For any non-empty DBM M, there exists a non-empty DBM N such that $\mathsf{Ball}_{d_\infty}(\nu, \varepsilon) \cap M \subseteq N$ for some $\nu \in M$, and for any shrinking matrix P with $(M, P) \neq \emptyset$, $N \subseteq (M, P)$. Moreover, N is computable in polynomial time.*

The last element we need for our proof is an observation from [BA11]: If π is a cycle of $\mathcal{R}(\mathcal{A})$ such that $\Gamma(\pi)$ is a complete graph, then for all $(\ell, \nu), (\ell, \nu') \in \mathsf{first}(\pi)$, there is a run $(\ell, \nu) \xrightarrow{\pi} (\ell, \nu')$, hence the reachability relation is complete.

Proof: Winning aperiodic lassos implies robust safety. We write $r = \mathsf{first}(\pi)$. Let $s \subseteq r$ given by Lemma 18 applied to r. Because $\Gamma(\pi)$ is complete, we have, by previous remark $r = \mathsf{CPre}_\pi^0(s)$. By Lemma 15, there exists a SM Q such that $(r, Q) = \mathsf{CPre}_\pi^\delta(s)$. By Lemma 16, (r, Q) is non-empty. By definition of s, for small enough $\delta > 0$, $s \subseteq (r, Q)$, so Controller has a strategy to always move inside s, at each iteration of π. Similarly, $\mathsf{CPre}_{\pi_0}^\delta((r, Q))$ is also non-empty and therefore contains the initial state. Hence, Controller wins.

Now, to actually compute δ and a winning strategy in exponential time, given an aperiodic lasso, one iterates the cycle so as to obtain a complete folded orbit graph (Lemma 17), then picks a subset s as in Lemma 18, and uses Corollary 15 to compute the controllable predecessors of s. This also provides the greatest δ under which the strategy along given lasso is valid.

7 Future Works

We intend to investigate possible extensions of this work to timed games, where Perturbator entirely determines the move in some locations. This could require generalizing the notion of aperiodicity from paths to trees since Controller can no more ensure to follow a given path. Another interesting future work is studying infinite runs in the semantics of [BMS12].

References

[AD94] Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)

[BA11] Basset, N., Asarin, E.: Thin and thick timed regular languages. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 113–128. Springer, Heidelberg (2011)

[BM83] Berthomieu, B., Menasche, M.: An enumerative approach for analyzing time Petri nets. In: WCC 1983, pp. 41–46. North-Holland/IFIP (1983)

[BMS12] Bouyer, P., Markey, N., Sankur, O.: Robust reachability in timed automata: A game-based approach. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part II. LNCS, vol. 7392, pp. 128–140. Springer, Heidelberg (2012)

[BY04] Bengtsson, J.E., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)

[CHP11] Chatterjee, K., Henzinger, T.A., Prabhu, V.S.: Timed parity games: Complexity and robustness. Logical Methods in Computer Science 7(4) (2011)

[CHR02] 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)

[DDMR08] 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)

[DDR05] 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)

[Dil90] 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)

[GHJ97] 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)

[Mar11] Markey, N.: Robustness in real-time systems. In: SIES 2011, Västerås, Sweden, pp. 28–34. IEEE Comp. Soc. Press (2011)

[Pur00] Puri, A.: Dynamical properties of timed automata. Discrete Event Dynamic Systems 10(1-2), 87–113 (2000)

[SBM11] Sankur, O., Bouyer, P., Markey, N.: Shrinking timed automata. In: FSTTCS 2011. LIPIcs, vol. 13, pp. 375–386. Leibniz-Zentrum für Informatik (2011)

[SBMR13] Sankur, O., Bouyer, P., Markey, N., Reynier, P.-A.: Robust controller synthesis in timed automata. Research Report LSV-13-08, Laboratoire Spécification et Vérification, ENS Cachan, France, 27 pages (2013)

[Sta12] Stainer, A.: Frequencies in forgetful timed automata. In: Jurdziński, M., Ničković, D. (eds.) FORMATS 2012. LNCS, vol. 7595, pp. 236–251. Springer, Heidelberg (2012)

Author Index