Philippa Gardner Nobuko Yoshida (Eds.)

NCS 3170

CONCUR 2004 – Concurrency Theory

15th International Conference London, UK, August/September 2004 Proceedings



Lecture Notes in Computer Science

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

Editorial Board

David Hutchison Lancaster University, UK Takeo Kanade Carnegie Mellon University, Pittsburgh, PA, USA Josef Kittler University of Surrey, Guildford, UK Jon M. Kleinberg Cornell University, Ithaca, NY, USA Friedemann Mattern ETH Zurich. Switzerland John C. Mitchell Stanford University, CA, USA Moni Naor Weizmann Institute of Science, Rehovot, Israel Oscar Nierstrasz University of Bern, Switzerland C. Pandu Rangan Indian Institute of Technology, Madras, India Bernhard Steffen University of Dortmund, Germany Madhu Sudan Massachusetts Institute of Technology, MA, USA Demetri Terzopoulos New York University, NY, USA Doug Tygar University of California, Berkeley, CA, USA Moshe Y. Vardi Rice University, Houston, TX, USA Gerhard Weikum Max-Planck Institute of Computer Science, Saarbruecken, Germany This page intentionally left blank

Philippa Gardner Nobuko Yoshida (Eds.)

CONCUR 2004 – Concurrency Theory

15th International Conference London, UK, August 31 – September 3, 2004 Proceedings



TERM LING

eBook ISBN: 3-540-28644-6 Print ISBN: 3-540-22940-X

©2005 Springer Science + Business Media, Inc.

Print ©2004 Springer-Verlag Berlin Heidelberg

All rights reserved

No part of this eBook may be reproduced or transmitted in any form or by any means, electronic, mechanical, recording, or otherwise, without written consent from the Publisher

Created in the United States of America

Visit Springer's eBookstore at: and the Springer Global Website Online at: http://ebooks.springerlink.com http://www.springeronline.com

Preface

This volume contains the proceedings of the 15th International Conference on Concurrency Theory (CONCUR 2004) held in the Royal Society, London, UK, from the 31st August to the 3rd September, 2004.

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

This volume starts with four invited papers from Sriram Rajamani, Steve Brookes, Bengt Jonsson and Peter O'Hearn. The remaining 29 papers were selected by the program committee from 134 submissions, a record number of submissions to CONCUR. The standard was extremely high and the selection difficult. Each submission received at least three reports, reviewed by the program committee members or their subreferees. Once the initial reviews were available, we had 16 days for paper selection and conflict resolution. We would like to thank all members of the CONCUR 2004 Program Committee for their excellent work throughout the intensive selection process, together with many subreferees who assisted us in the evaluation of the submitted papers.

The conference includes talks by several invited speakers: invited seminars by David Harel (Weizmann Institute) and Sriram Rajamani (Microsoft Research, Redmond), and invited tutorials by Steve Brooks (Carnegie-Mellon) and Peter O'Hearn (Queen Mary, University of London), and by Bengt Jonsson (Uppsala).

The conference has 11 satellite events:

- Workshop on Structural Operational Semantics (SOS 2004), organised by Luca Aceto.
- 11th International Workshop on Expressiveness in Concurrency (EXPRESS 2004), organised by Flavio Corradini.
- II Workshop on Object-Oriented Developments (WOOD 2004), organised by Viviana Bono.
- 3rd International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA 2004), organised by Jean-Marie Jacquet.
- 2nd International Workshop on Security Issues in Coordination Models, Languages and Systems (SECCO 2004), organised by Gianluigi Zavattaro.

- Workshop on Concurrent Models in Molecular Biology (BIOCONCUR 2004), organised by Anna Ingolfsdottir.
- Global Ubiquitous Computing (FGUC 2004), organised by Julian Rathke.
- 3rd International Workshop on Parallel and Distributed Methods in Verification (PDMC 2004), organised by Martin Leucker.
- 4th International Workshop on Automated Verification of Critical Systems (AVoCS 2004), organised by Michael Huth.
- 1st International Workshop on Practical Applications of Stochastic Modelling (PASM 2004), organised by Jeremy Bradley.
- 6th International Workshop on Verification of Infinite-State Systems (INFINITY 2004), organised by Julian Bradfield.

We would like to thank the conference organisation chair Iain Phillips, the local organisers Alex Ahern and Sergio Maffeis, the workshop organisation chairs Julian Rathke and Vladimiro Sassone, and the workshop organisers. Finally we thank the invited speakers, invited tutorial speakers and the authors of submitted papers for participating in what promises to be a very interesting conference.

We gratefully acknowledge support from the Department of Computing, Imperial College London, the Engineering and Physical Sciences Research Council (EPSRC), Microsoft Research in Cambridge, and the Royal Society.

June 2004

Philippa Gardner and Nobuko Yoshida

Organisation

CONCUR Steering Committee

Roberto Amadio (Université de Provence, Marseille, France) Jos Baeten (Technische Universiteit Eindhoven, Netherlands) Eike Best (Carl von Ossietzky Universität Oldenburg, Germany) Kim G. Larsen (Aalborg University, Denmark) Ugo Montanari (Università di Pisa, Italy) Scott Smolka (State University of New York at Stony Brook, USA) Pierre Wolper (Université de Liège, Belgium)

Program Committee

Luca Aceto (Aalborg University, Denmark) Luca de Alfaro (University of California, Santa Cruz, USA) Bruno Blanchet (Saarbrücken, Germany/École Normale Supérieure, France) Steve Brookes (Carnegie Mellon University, USA) Philippa Gardner (Imperial College London, UK, Co-chair) Paul Gastin (Université Paris 7, France) Petr Jančar (Technical University of Ostrava, Czech Republic) Joost-Pieter Katoen (University of Twente, Netherlands) Dietrich Kuske (Technische Universität Dresden, Germany) Cosimo Laneve (Università di Bologna, Italy) Michael Mendler (Otto-Friedrich-Universität Bamberg, Germany) Ugo Montanari (Università di Pisa, Italy) Catuscia Palamidessi (INRIA Futurs Saclay and LIX, France) Vladimiro Sassone (University of Sussex, UK) PS Thiagarajan (National University of Singapore, Singapore) Antti Valmari (Tampere University of Technology, Finland) Wang Yi (Uppsala Universitet, Sweden) Nobuko Yoshida (Imperial College London, UK, Co-chair)

Referees

Parosh Abdulla Joaquin Aguado Rajeev Alur Roberto Amadio Christopher Anderson Suzana Andova Christel Baier Michael Baldamus Paolo Baldan Howard Barringer Gerd Behrmann Nick Benton Béatrice Bérard Martin Berger Marco Bernardo Bernard Berthomieu Bernard Boigelot Roland Bol Frank de Boer Michele Boreale Ahmed Bouajjani Patricia Bouyer Julian Bradfield Ed Brinksma Roberto Bruni Glenn Bruns Nadia Busi Cristiano Calcagno Luca Cardelli Josep Carmona Samuele Carpineti Iliano Cervesato Tom Chothia Giovanni Conforti Andrea Corradini Véronique Cortier Dennis Dams Philippe Darondeau Alexandre David Jennifer Davoren Conrado Daws Giorgio Delzanno Stephane Demri Henning Dierks Alessandra Di Pierro Dino Distefano Marco Faella Lisbeth Fajstrup Alessandro Fantechi Jérôme Feret Gianluigi Ferrari Elena Fersman Emmanuel Fleury Riccardo Focardi Marcelo Fiore Wan Fokkink Cédric Fournet Laurent Fribourg Murdoch Gabbay Maurizio Gabbrielli Fabio Gadducci

Simon Gay Jaco Geldenhuvs Neil Ghani Giorgio Ghelli Paola Giannini Gregor Goessler Goerges Gonthier Daniele Gorla Olga Greinchtein Dilian Gurov Elsa Gunter Peter Habermehl John Hakansson Henri Hansen Jens Hansen Zdenek Hanzalek Thomas Hildebrandt Daniel Hirschkoff Kohei Honda Michael Huth Hans Hüttel Radha Jagadeesan Somesh Jha Ranjit Jhala Bengt Jonsson Gabriel Juhas Marcin Jurdzinski Antero Kangas Antti Kervinen Mike Kishinevsky Bartek Klin Barbara König Juha Kortelainen Martin Kot Pavel Krčál Mojmír Křetínský Stephan Kreutzer Lars Kristensen Kare Kristoffersen Antonín Kučera Narayan Kumar Orna Kupferman Marta Kwiatkowska Jim Laird Ivan Lanese Martin Lange

Rom Langerak Francois Laroussinie Kim Larsen Paola Lecca Fabrice Le Fessant James Leifer Francesca Levi Kamal Lodava Markus Lohrey Robert Lorenz Gerald Luettgen Markus Lumpe Damiano Macedonio Angelika Mader Sergio Maffeis A. Maggiolo-Schettini Pritha Mahata Jean Mairesse Rupak Majumdar Andreas Maletti Fabio Martinelli Narciso Marti-Oliet Angelika Mader Richard Mayr Hernan Melgratti Oskar Mencer Massimo Merro Marino Miculan Dale Miller Robin Milner Sebastian Moedersheim Faron Moller Angelo Montanari Rémi Morin Larry Moss Madhavan Mukund Andrzej Murawski Anca Muscholl Uwe Nestmann Peter Niebert Mogens Nielsen Barry Norton Robert Palmer Jun Pang Matthew Parkinson Justin Pearson

Christian Pech Paul Pettersson Iain Phillips Michele Pinna Marco Pistore Lucia Pomello Franck Pommereau François Pottier John Power Rosario Pugliese Antti Puhakka Alexander Rabinovich R. Ramanujam Julian Rathke Martin Raussen Wolfgang Reisig Michel Reniers Arend Rensink James Riely Grigore Rosu Olivier Roux Abhik Roychoudhury

Ernst Ruediger Olderog Theo Ruys Ugo de Ruvs Claudio Sacerdoti Coen Davide Sangiorgi Zdenek Sawa Karsten Schmidt Alan Schmitt Philippe Schnoebelen Carsten Schuermann Thomas Schwentick Stefan Schwoon Christof Simons Riccardo Sisto Pawel Sobocinski Chin Soon Lee Jiri Srba Till Tantau Jan Tertmans Simone Tini Alwen Tiu Mikko Tiusanen

Irek Ulidowski Yaroslav Usenko Frank Valencia Frank van Breugel Jaco van de Pol Francois Vernadat Bjorn Victor Maria Vigliotti Walter Vogler Igor Walukiewicz Pascal Weil Lisa Wells Michael Westergaard Thomas Wilke Tim Willemse Jozef Winkowski Guido Wirtz Lucian Wischik Hongseok Yang Gianluigi Zavattaro Marc Zeitoun

This page intentionally left blank

Table of Contents

Invited Papers

Zing: Exploiting Program Structure for Model Checking Concurrent Software	
Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, Yichen Xie	1
A Semantics for Concurrent Separation Logic Stephen Brookes	16
A Survey of Regular Model Checking Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, Mayank Saksena	35
Resources, Concurrency and Local Reasoning Peter W. O'Hearn	49
Accepted Papers	
Resource Control for Synchronous Cooperative Threads Roberto M. Amadio, Silvano Dal Zilio	68
Verifying Finite-State Graph Grammars: An Unfolding-Based Approach Paolo Baldan, Andrea Corradini, Barbara König	83

The Pros and Cons of Netcharts Nicolas Baudru, Rémi Morin	99
Basic Theory of Reduction Congruence for Two Timed Asynchronous π-Calculi Martin Berger	115
Characterizing EF and EX Tree Logics Mikołaj Bojańczyk, Igor Walukiewicz	131
Message-Passing Automata Are Expressively Equivalent to EMSO Logic Benedikt Bollig, Martin Leucker	146
Symbolic Bisimulation in the Spi Calculus Johannes Borgström, Sébastien Briais, Uwe Nestmann	161
A Symbolic Decision Procedure for Cryptographic Protocols with Time Stamps	

Liana Bozga, Cristian Ene, Yassine Lakhnech 177

Deciding Probabilistic Bisimilarity Over Infinite-State Probabilistic Systems <i>Tomáš Brázdil, Antonín Kučera, Oldřich Stražovský</i>	193
μABC: A Minimal Aspect Calculus Glenn Bruns, Radha Jagadeesan, Alan Jeffery, James Riely	209
Type Based Discretionary Access Control Michele Bugliesi, Dario Colazzo, Silvia Crafa	225
Elimination of Quantifiers and Undecidability in Spatial Logics for Concurrency <i>Luís Caires, Étienne Lozes</i>	240
Modular Construction of Modal Logics Corina Cîrstea, Dirk Pattinson	258
Verification by Network Decomposition Edmund Clarke, Muralidhar Talupur, Tayssir Touili, Helmut Veith	276
Reversible Communicating Systems Vincent Danos, Jean Krivine	292
Parameterised Boolean Equation Systems Jan Friso Groote, Tim Willemse	308
An Extensional Spatial Logic for Mobile Processes Daniel Hirschkoff	325
Timed vs. Time-Triggered Automata Pavel Krčál, Leonid Mokrushin, P.S. Thiagarajan, Wang Yi	340
Extended Process Rewrite Systems: Expressiveness and Reachability Mojmír Křetínský, Vojtěch Řehák, Jan Strejček	355
A General Approach to Comparing Infinite-State Systems with Their Finite-State Specifications	371
Model Checking Timed Automata with One or Two Clocks <i>F. Laroussinie, N. Markev, Ph. Schnoebelen</i>	387
On Flatness for 2-Dimensional Vector Addition Systems with States Jérôme Leroux, Grégoire Sutre	402
Compiling Pattern Matching in Join-Patterns Qin Ma, Luc Maranget	417
Model Checking Restricted Sets of Timed Paths Nicolas Markey, Jean-François Raskin	432

Asynchronous Games 2: The True Concurrency of Innocence Paul-André Melliès	448
Open Maps, Alternating Simulations and Control Synthesis Paulo Tabuada	466
Probabilistic Event Structures and Domains Daniele Varacca, Hagen Völzer, Glynn Winskel	481
Session Types for Functional Multithreading Vasco Vasconcelos, António Ravara, Simon Gay	497
A Higher Order Modal Fixed Point Logic Mahesh Viswanathan, Ramesh Viswanathan	512
Author Index	529

This page intentionally left blank

Zing: Exploiting Program Structure for Model Checking Concurrent Software

Tony Andrews*, Shaz Qadeer*, Sriram K. Rajamani*, Jakob Rehof*, and Yichen Xie[†]

*Microsoft Research [†] Stanford University http://www.research.microsoft.com/zing/

Abstract. Model checking is a technique for finding bugs in systems by systematically exploring their state spaces. We wish to extract sound models from concurrent programs automatically and check the behaviors of these models systematically. The ZING project is an effort to build a flexible infrastructure to represent and model check abstractions of large concurrent software.

To support automatic extraction of models from programs written in common programming languages, ZING's modeling language supports three facilities present in modern programming languages: (1) procedure calls with a call-stack, (2) objects with dynamic allocation, and (3) processes with dynamic creation, using both shared memory and message passing for communication. We believe that these three facilities capture the essence of model checking modern concurrent software.

Building a scalable model-checker for such an expressive modeling language is a huge challenge. ZING's modular architecture provides a clear separation between the expressive semantics of the modeling language, and a simple view of ZING programs as labeled transition systems. This separation has allowed us to decouple the design of efficient model checking algorithms from the complexity of supporting rich constructs in the modeling language.

ZING's model checking algorithms have been designed to exploit existing structural abstractions in concurrent programs such as processes and procedure calls. We present two such novel techniques in the paper: (1) compositional checking of ZING models for message-passing programs using a conformance theory inspired by work in the process algebra community, and (2) a new summarization algorithm, which enables ZING to reuse work at procedure boundaries by extending interprocedural dataflow analysis algorithms from the compiler community to analyze concurrent programs.

1 Introduction

The goal of the ZING project is to check properties of concurrent heapmanipulating programs using model checking. By systematically exploring the state space, model checkers are able to find tricky concurrency errors that are impossible to find using conventional testing methods. Industrial software has such large number of states and it is infeasible for any systematic approach to cover all the reachable states. Our goal is to automatically extract a *model* from a program, where a model keeps track of only a small amount of information about the program with respect to the property being checked. Then, it is feasible to systematically explore all the states of the model. Further, we want these models to be sound abstractions of the program —a property proved on the model should hold on the program as well.

How expressive should the model be? Choosing a very restricted model such as finite-state machines makes the task of building the model checker easy, but the task of extracting such a model from a program becomes hard. On the other hand, building a model checker directly for a programming language is hard, due to the number of features present in programming languages. We believe that the following features capture the essence of modern concurrent object oriented languages, from the point of building sound abstractions for model checking: (1) procedure calls with a call-stack, (2) objects with dynamic allocation, and (3) processes with dynamic creation, using both shared memory and message passing for communication. We designed ZING's modeling language to have exactly these features.

Building a scalable model checker for the ZING modeling language is a huge challenge since the states of a ZING model have complicated features such as processes, heap and stack. We designed a lower-level model called ZING object model (or ZOM), and built a ZING compiler to convert a ZING model to ZOM. The compiler provides a clear separation between the expressive semantics of the modeling language, and a simple view of ZOM as labeled transition systems. This separation has allowed us to decouple the design of efficient model checking algorithms from the complexity of supporting rich constructs in the modeling language.

Writing a simple DFS model checker on top of ZOM is very easy and can be done with a 10-line loop. However, this simple model checker does not scale. For building scalable checkers, we have to exploit the structural boundaries present in the source program that are preserved in the ZING model. Processes, procedures and objects are perhaps the structural abstractions most widely used by programmers. Structural boundaries enable compositional model checking, and help alleviate the state-explosion problem. For implementing optimized model checking algorithms that exploit such structure, we had to expose more information about the state of the model in ZOM.

In well-synchronized shared memory programs, any computation of a process can be viewed as a sequence of transactions, each of which appears to execute atomically to other processes. An action is called a right (left) mover if it can be committed to the right (left) of any action of another process in any execution. A transaction is a sequence of right movers, followed by at most a single atomic action, followed by a sequence of left movers. During model checking, it is sufficient to schedule processes only at transaction boundaries, and this results in an exponential reduction in the number of states explored. To implement such transaction-based reduction, we extended the ZOM to expose information about the type of action executed —right mover, left mover, both left and right mover, neither left nor right mover.

The ability to summarize procedures is fundamental to building scalable interprocedural analyses. For sequential programs, procedure summarization is well-understood and used routinely in a variety of compiler optimizations and software defect-detection tools. This is not the case for concurrent programs. If we expose procedure boundaries in the ZOM, we can summarize procedures that are entirely contained within transactions. When a transaction starts in one procedure and ends in another, we can break the summary piece-wise and record smaller sub-summaries in the context of each sub-procedure. The procedure summaries thus computed allow reuse of analysis results across different call sites in a concurrent program, a benefit that has hitherto been available only to sequential programs [15].

We are interested in checking that a process in a communicating system cannot wait indefinitely for a message that is never sent, and cannot send a message that is never received. A process that passes this check is said to be *stuck-free* [16,7,8]. We have defined a conformance relation \leq on processes with the following substitutability property: If $I \leq C$ and P is any environment such that the parallel composition $P \mid C$ is stuck-free, then $P \mid I$ is stuck-free as well. Substitutability enables a component's specification to be used instead of the component in invocation contexts, and hence enables model checking to scale. By exposing observable events during the execution of each action in ZOM, we can build a conformance-checker to check if one ZING model (the implementation) conforms with another ZING model (the specification).

The goal of this paper is to describe the architecture and algorithms in ZING. A checking tool is useless without compelling applications where the checker provides value. We have used ZING to check stuck-freeness of distributed applications, concurrency errors in device drivers, and protocol errors in a replicated file system. We have also built extractors from several programming languages to ZING. Since ZING provides core features of object-oriented languages, building such extractors is conceptually simple. Describing the details of these applications and extractors is beyond the scope of this paper.

To summarize, the ZING project is centered around three core principles:

- 1. It is possible to extract sound models from concurrent programs. To enable construction of simple extractors from common programming languages, the ZING modeling language has three core features (1) procedure calls, (2) objects and (3) processes.
- 2. It is beneficial to construct an intermediate model ZOM, which presents a simple view of ZING models as labeled transition systems. We have constructed various model checkers over this simple view.
- 3. Since ZING's modeling language preserves abstraction boundaries in the source program, we can exploit these boundaries to do compositional model checking, and help alleviate the state-explosion problem. Doing this requires exposing more information about the state and actions in ZOM. By exposing mover information about executed actions we have been able to imple-

ment transaction based reduction. By exposing information about procedure boundaries, we have been able to implement a novel summarization algorithm for concurrent programs. By exposing the observable events during execution of each action, we have been able to build a novel conformance checker to compositionally check if a ZING model is stuck-free.

Related Work. The SPIN project [10] pioneered explicit-state model checking of concurrent processes. The SPIN checker analyzes protocol-descriptions written in the PROMELA language. Though PROMELA supports dynamic process creation, it is difficult to encode concurrent software in PROMELA due to absence of procedure calls and objects. Efforts have been made to abstract C code into PROMELA [11] to successfully find several bugs in real-life telephone switching systems, though no guarantees were given as to whether the generated PROMELA model is a sound abstraction of the C code. Over the past few years, there has been interest in using SPIN-like techniques to model check software written in common programming languages. DSPIN was an effort to extend SPIN with dynamic software-like constructs [12]. Model checkers have also been written to check Java programs either directly [21, 20, 18] or by constructing slices or other abstractions [6]. Unlike ZING none of these approaches exploit program abstractions such as processes and procedure calls to do modular model checking. The SLAM project [4] has similar goals to ZING in that it works by extracting sound models from C programs, and checking the models. SLAM has been very successful in checking control-dominated properties of device drivers written in C. Unlike ZING, it does not handle concurrent programs, and it is unable to prove interesting properties on heap-intensive programs.

Outline. The remainder of the paper is structured as follows. We explain the features of ZING's modeling language, and discuss the modular software architecture of ZING in Section 2. We discuss the novel compositional algorithms of ZING in Section 3. Section 4 concludes the paper with a discussion of current status and future work.

2 Architecture

ZING's modeling language provides several features to support automatic generation of models from programs written in common programming languages. It supports a basic asynchronous interleaving model of concurrency with both shared memory and message passing. In addition to sequential flow, branching and iteration, ZING supports function calls and exception handling. New processes are created via asynchronous function calls. An asynchronous call returns to the caller immediately, and the callee runs as a fresh process in parallel with the caller. Primitive and reference types, and an object model similar to C# or Java is supported, although inheritance is currently not supported. ZING also provides features to support abstraction and efficient state exploration. Any sequence of statements (with some restrictions) can be bracketed as atomic. This is essentially a directive to the model checker to not consider interleavings with

5

other processes while any given process executes an atomic sequence. Sets are supported, to represent collections where the ordering of objects is not important (thus reducing the number of potentially distinct states ZING needs to explore). A choose construct that can be used to non-deterministically pick an element out of a finite set of integers, enumeration values, or object references is provided. A complete language specification can be found in [1]. An example ZING model that we extracted from a device driver, and details of an error trace that the ZING model checker found in the model can be found in [2].



Fig. 1. Architecture of ZING

ZING is designed to have flexible software architecture. The architecture is designed to promote an efficient division of labor between model checking researchers and domain experts, and make it possible for model checking researchers to innovate in the core state-space exploration technology while allowing domain-experts to tackle issues such as extracting ZING models from their source code, and visualization for showing results from the model checker. Once model extraction is done, the generated ZING model is fed into a ZING compiler which converts the ZING model into an MSIL¹ object code called ZING object model (ZOM). The object code supports a specific interface intended to be used by the model checker. The ZOM assembly has an object of type *State* which has a

¹ MSIL stands for Microsoft Intermediate Language which is the instruction set for Microsoft's Common Language Runtime.

stack for each process, a global storage area of static class variables, and a heap for dynamically allocated objects. Several aspects of managing the internals of the State object can be done generically, for all ZING models. This common state management functionality is factored into a the ZING runtime library.

The equality operator for the State class is overridden to test equality using a "fingerprint" of the state with the following property: (1) If state s_1 is a symmetric equivalent of state s_2 then $fingerprint(s_1) = fingerprint(s_2)$, and (2) If $fingerprint(s_1) = fingerprint(s_2)$, then states s_1 and s_2 are equivalent with a high probability. Because states are compared frequently and the state vector is potentially large, the use of fingerprints is generally advantageous. Further, when all of the immediate children of a state have been generated, the full state representation may be discarded provided the fingerprint is retained. Two states are equivalent if the contents of the stacks and global variables are *identical* and the heaps are *isomorphic*. The fingerprinting algorithm for the State object first constructs a canonical representation of the state by traversing the heap in a deterministic order [12]. Thus, equivalent states have equal fingerprints. We observe that most state transitions modify only a small portion of the State object. The State object records an "undo-log" and uses it to reverse transitions, thereby avoiding cloning the entire state while doing depth-first search.

```
Stack dfsStack;
Hashtable stateHash;
void addState(State I) {
    if (!stateHash.Contains(I)) {
         stateHash.Add(I);
         dfsStack.Push(I);
    }
}
void doDfs(State initialState) {
    addState(initialImplState);
    while (dfsStack.Count \geq 1) {
         State I = (State) dfsStack.Peek();
         State newI = I.GetNextSuccessor();
         if (newI != null)
              addState(newI);
         else
              dfsStack.Pop();
    }
}
```

Fig. 2. Simple DFS model checker for ZING

The State object exposes a GetNextSuccessor method that returns the next successor of the state. By iteratively calling this method, all successor states of the current state can be generated. Model checkers use the method GetNextSuccessor to execute a process for one atomic step. The execution semantics of the process, which includes complicated activities like process creation, function call, exceptions, dynamic memory allocation, are all handled by the implementation of GetNextSuccessor using support from the ZING compiler and runtime. Model checkers are thus decoupled from the intricate execution semantics supported by ZING. The actual implementation of the State object is

7

quite complicated since it has to represent stacks for each process, a global area and the heap. Using the interface provided by ZOM's State object, a simple depth-first search model checker for ZING can be written in less than ten lines as shown in Figure 2. The model checker stores finger prints of visited states in a hash table stateHash. When visiting each new state, the model checker first checks if the fingerprint of the new state is already present in the stateHash, and if present avoids re-exploring the new state. When the checker reaches an erroneous state, the entire trace that leads to the error is present in the model checker's DFS stack, and we can display the trace at the source level (this is omitted in Figure 2 for simplicity).

3 Algorithms

Since ZING's modeling language preserves abstraction boundaries in the source program, we can exploit these boundaries to do compositional model checking, and help alleviate the state-explosion problem. Doing this requires exposing more information about the state and actions in ZOM. By exposing mover information about executed actions, we have been able to implement transaction based reduction. By exposing information about procedure boundaries, we have been able to implement a novel summarization algorithm for concurrent programs. By exposing the observable events during execution of each action, we have been able to build a novel conformance checker to compositionally check if a ZING model is stuck-free.

3.1 Model Checker with Reduction

We have implemented a state-reduction algorithm that has the potential to reduce the number of explored states exponentially without missing errors. This algorithm is based on Lipton's theory of reduction [13]. Our algorithm is based on the insight that in well-synchronized programs, any computation of a process can be viewed as a sequence of transactions, each of which appears to execute atomically to other processes. An action is called a right mover if can be commuted to the right of any action of another process in any execution. Similarly, an action is called a *left mover* if can be commuted to the left of any action of another process in any execution. A transaction is a sequence of right movers, followed by a single (atomic) action with no restrictions, followed by a sequence of left movers. During state exploration, it is sufficient to schedule processes only at transaction boundaries. These inferred transactions reduce the number of interleavings to be explored, and thereby greatly alleviate the problem of state explosion. To implement transaction-based reduction, we augmented the GetNextSuccessor method so that it returns the type of the action executed (i.e., left mover, right mover, non mover or both mover), and the model checker uses this information to infer transaction boundaries.

3.2 Model Checker with Summarization

The ability to summarize procedures is fundamental to building scalable interprocedural analyses. For sequential programs, procedure summarization is well-understood and used routinely in a variety of compiler optimizations and software defect-detection tools. The *summary* of a procedure P contains the state pair (s, s') if in state s, there is an invocation of P that yields the state s' on termination. Summaries enable reuse—if P is called from two different places with the same state s, the work done in analyzing the first call is reused for the second. This reuse is the key to scalability of interprocedural analyses. Additionally, summarization avoids direct representation of the call stack, and guarantees termination of the analysis even if the program has recursion.

However, the benefit of summarization is not available to concurrent programs, for which a clear notion of summaries has so far remained unarticulated in the research literature. ZING has a novel two-level model checking algorithm for concurrent programs using summaries [15]. The first level performs reachability analysis and maintains an explicit stack for each process. The second level computes a summary for each procedure. During the reachability analysis at the first level, whenever a process makes a procedure call, we invoke the second level to compute a summary for the procedure. This summary is returned to the first level, which uses it to continue the reachability analysis. The most crucial aspect of this algorithm is the notion of procedure summaries in concurrent programs. A straightforward generalization of a (sequential) procedure summary to the case of concurrent programs could attempt to accumulate all state pairs (s, s')obtained by invoking this procedure in any process. But this simple-minded extension is not that meaningful, since the resulting state s' for an invocation of a procedure P in a process might reflect updates by interleaved actions of concurrently executing processes. Clearly, these interleaved actions may depend on the local states of the other processes. Thus, if (s, s') is an element of such a summary, and the procedure P is invoked again by some process in state s, there is no guarantee that the invoking process will be in state s' on completing execution of P. However, in well-synchronized programs, any computation of a process can be viewed as a sequence of transactions, each of which appears to execute atomically to other processes. Thus, within a transaction, we are free to summarize procedures. Two main technical difficulties arise while performing transaction-based summarization of procedures:

- Transaction boundaries may not coincide with procedure boundaries. One way to summarize such transactions is to have a stack frame as part of the state in each summary. However, this solution not only complicates the algorithm but also makes the summaries unbounded even if all state variables have a finite domain. Our summaries do *not* contain stack frames. If a transaction begins in one procedure context and ends in another procedure context, we break up the summary into smaller sub-summaries each within the context of a single procedure. Thus, our model checking algorithm uses a combination of two representations—states with stacks and summaries without stacks.

- A procedure can be called from different phases of a transaction —the precommit phase or the post-commit phase. We need to summarize the procedure differently depending on the phase of the transaction at the call site. We solve this problem by instrumenting the source program with a boolean variable representing the transaction phase, thus making the transaction phase part of the summaries.

Assertion checking for concurrent programs with finite-domain variables and recursive procedures is undecidable [17]. Thus, the two-level model-checking algorithm is not guaranteed to terminate. However, if all variables are finite domain and every call to a recursive procedure is contained entirely within a transaction, the two-level algorithm will terminate with the correct answer [15].

```
int g;
int baz(int x, int y){
  g = x+1;
}
```

Fig. 3. Small example to illustrate patterns and effects

Our implementation of the two-level model checking algorithm in ZING represents a summary as a *pattern* and *effect* pair, rather than a state pair. A pattern is a partial map from (read) variables to values, and an effect is a partial map from (written) variables to values. The ZOM supports summarization by exposing (1) whether the executed action is a procedure call or return, and (2) what variables are read and written during an action. Patterns and effects enable better reuse of summaries than state pairs. For example, consider the function baz from Figure 3. If baz is called with a state (x=0, y=1, g=0), it results in state (x=0, y=1, g=1). We represent a summary of this computation as a pattern (x=0) and an effect (g=1). Thus, if baz is called with a state (x=0, y=10, g=3), it still matches the pattern (x=0), and the effect (g=1) can be used to compute the resulting state (x=0, y=10, g=1). In contrast, if the summary is represented as a state pair ((x=0, y=1, g=0), (x=0, y=1, g=1)), then the summary cannot be reused if baz were called at state (x=0, y=10, g=3).

The model checker BEBOP [3] from the SLAM project represents summaries as state pairs. In order to illustrate the efficiency of reuse we present empirical comparison between ZING's implementation of summarization and BEBOP's implementation. Since BEBOP supports model checking of sequential programs only, we do the comparison with a parameterized set of sequential ZING models shown in Figure 4. Program P(n) contains n global boolean variables g1, g2, ..., gn and n procedures level1, level2, ..., leveln. Figure 5 shows the running times for ZING and BEBOP for models $P(10), P(20), \ldots, P(100)$. Due to the use of patterns and effects for representing summaries, the ZING runtime for P(n) scales linearly with n.

```
10
       T. Andrews et al.
class BoolProg {
  static bool g1;
  static bool g2;
  . . .
  static bool g<n>;
  activate static void main() {
    level1(true, true, true);
    level1(true, true, true);
  }
  static void level<i>(bool p1, bool p2, bool p3) {
    bool a,b,c;
    a = false;b = false;c = false;
    while(!a|!b|!c) {
      if (!a)
        a = true ;
      else if (!b)
        {a = false; b = true;}
      else if (!c)
        {a = false; b = false; c = true;}
      g<i> = false;
      level<i+1>(a, b, c);
      g<i> = true;
      level<i+1>(a, b, c);
      g<i> = false;
   }}}
```

Fig. 4. Template to evaluate summary reuse using patterns and effects

3.3 Conformance Checker

We are interested in checking that a ZING process cannot get into a state where it waits for messages that are never sent (deadlock) or has sent messages that are never received (orphan messages, for example, unhandled exception messages). We say, informally, that a processes is *stuck* if it cannot make any transition whatsoever, and yet some component of it is ready to send or receive a message. We say that a process is *stuck-free*, if it cannot transition to a stuck state.²

In order to check for stuck-freedom compositionally (one component at a time) for a system of communicating processes, we have defined a refinement relation \leq , called stuck-free conformance, which allows us to regard one ZING process as a specification of another. Stuck-free conformance is a simulation relation on ZING processes, which (*i*) is preserved by all contexts and (*ii*) preserves the ability to get stuck. From these properties it follows that, if P and Q are ZING processes such that $P \leq Q$, then for any process R, if $R \mid Q$ is stuck-free, then $R \mid P$ is stuck-free ($P \mid Q$ denotes the parallel composition of P and Q,

² We have formalized the notion of stuckness and stuck-freedom for transition systems in CCS [14], and we refer to [8, 7] for the precise definition of stuck-free CCS processes.

11

ZING Vs BEBOP model checking times



Fig. 5. Runtimes for ZING and BEBOP on models from Figure 4

which is expressed in ZING via a sync calls.) Therefore, if $P \leq Q$, we can safely substitute Q (a specification) for P (an implementation) in any context when reasoning about stuck-freedom, thereby enabling compositional checking. Our definition of stuck-free conformance [8, 7] between ZING processes is the largest relation < such that, whenever P < Q, then the following conditions hold:

C1. If $P \xrightarrow{\tau^* \lambda} P'$ then there exists Q' such that $Q \xrightarrow{\tau^* \lambda} Q'$ and $P' \leq Q'$. C2. If *P* can refuse *X* while ready on *Y*, then *Q* can refuse *X* while ready on *Y*.

Here, $P \xrightarrow{\tau^* \lambda} P'$ means that P can transition to P' on a sequence of hidden actions, τ , and a visible action, λ . A process is called *stable*, if it cannot do any τ -actions. If X and Y are sets of visible actions, we say that P can refuse X while ready on Y, if there exists a stable P' such that $P \xrightarrow{\tau^*} P'$ and (i) P' refuses X, i.e., P' cannot do a co-action of any action in X, and (ii) P' is ready on Y, i.e., P' can do every action in Y. In condition [C2] above, the ready sets Y range only over singleton sets or the empty set. This requirement on Y leads to the most permissive simulation satisfying the preservation properties (i) and (*ii*) mentioned above.³

We have extended the ZOM interface so that we can observe externally visible actions as well as the occurrence of hidden actions:

³ Our notion of stuck-free conformance can be seen as a restriction of the natural simulation-based version of CSP stable failures refinement [5,9,19], which in addition to preserving deadlock also preserves the ability to generate orphan messages. We refer to [8, 7] for more details on the theory of stuck-free conformance.

```
Stack dfsStack;
Hashtable stateHash;
void addState(State I, State S) {
    StatePair combinedState = new StatePair(newI, newS);
    if (!stateHash.Contains(combinedState)) {
         stateHash.Add(combinedState);
         dfsStack.Push(combinedState);
    }
}
void checkConformance(State initialImplState, State initialSpecState) {
    addState(initialImplState, initialSpecState);
    while (dfsStack.Count \geq 1) {
         StatePair P = (StatePair) dfsStack.Peek();
         State I = P.first();
         State S = P.second();
         State newI = I.GetNextSuccessor();
         if (newI == null).
             if (isStable(I)) {
                  // First get all the events we executed from I.
                  ExternalEvent[] IEvents = I.AccumulatedExternalEvents;
                  // Check if ready-refusals of I are ready-refused by S as well.
                  for(int i = 0; i < IEvents.Count; i++)
                      if(!checkReadyRefusals(S, IEvents, IEvents[i])) {
                           Console.WriteLine("Ready refusals do not match up");
                           return:
                  }
             dfsStack.Pop();
             continue;
         ExternalEvent event = newI.ExternalEvent;
         // Try to produce a transition from newS with "event" as the observable event.
         State newS = executeWithEvent(S, event);
         if (newS == null) {
             Console.WriteLine("Implementation has unspecified behavior");
             return:
         addState(newI, newS);
    Console.WriteLine("I conforms with S");
}
```

Fig. 6. Conformance checker for ZING

- 1. ExternalEvent is a property which, for a newly generated state, gives the event (if any) on the transition that was used to generate the state.
- 2. AccumulatedExternalEvents gives an array of events from all outgoing transitions on a state, once all the outgoing transitions have been explored.

An implementation of the conformance checker using this interface is given in Figure 6. By exploring the state spaces of a given process P and a specification process C, checkConformance(P, C) decides whether $P \leq C$, by a direct implementation of conditions [C1] and [C2]. We assume that the specification does not have hidden nondeterminism. i.e., for a specification state S, if $S \xrightarrow{\tau^* \cdot \lambda} S_1$ and $S \xrightarrow{\tau^* \cdot \lambda} S_2$, then $S_1 \equiv S_2$. This assumption can be relaxed by determinizing the specification in a pre-processing step, or on-the fly using a subset construction. The conformance checker works by doing a depth-first-search on the state-space

of the implementation, and tracking the "matching" state of the specification corresponding to each state of the implementation. A hashtable is used to keep track of states that have been already visited. In our implementation, we store fingerprints of the visited states in the hashtables for efficiency. At each transition explored in the implementation, the algorithm checks for conditions [C1]. After all the successors of an implementation state have been explored, it is popped from the DFS stack. At that time, the algorithm checks if condition [C2] holds. The algorithm uses three functions executeWithEvent, isStable, and checkReadyRefusals. The function executeWithEvent searches the specification for a state which can be obtained by transitioning through the given event. Formally, executeWithEvent (S, λ) returns a state S' such that $S \xrightarrow{\tau^* \lambda} S'$ if such a state S' exists (note that such a state is unique if it exists due to the assumption that the specification does not have hidden nondeterminism). If this function returns null, then we conclude that condition [C1] has been violated. The function isStable returns TRUE if the given state S is stable and FALSE otherwise. The function checkReadyRefusals (S, X, λ) returns true if condition [C2] holds. More precisely, checkReadyRefusals (S, X, λ) returns TRUE if there exists a stable S' such that (i) $S \xrightarrow{\tau *} S'$ and (ii) for all λ' if $Q' \xrightarrow{\lambda'}$ then $\lambda' \in X$, and (*iii*) $S' \xrightarrow{\lambda}$. The algorithm terminates if the state space of the implementation is finite, and the complexity is linear in the state spaces of the implementation and the specification. If the state space of the implementation is too large or infinite, the algorithm can be used to check for conformance in whatever portion of the state space is explored.

4 Conclusion

The goal of the ZING project is to check properties of concurrent programs that manipulate the heap, by using natural abstraction boundaries that exist in the program. In order to support this goal, the ZING modeling language supports the essential features of modern object oriented languages, and the ZING architecture enables a clear separation between the expressiveness of the modeling language and the simplicity of the ZING object model (ZOM). This separation has enabled us to implement several novel model checking algorithms on top of the ZOM. We are currently implementing a few additional algorithms to enable ZING to scale to larger models:

- Currently non-determinism in data (introduced by the choose statement) is handled by an explicit case-split. We have designed a technique to handle such non-determinism symbolically. Our proposed algorithm adds symbolic fix-point computing capability to ZING, with the possibility of using widening to accelerate convergence.
- We are currently investigating how to design a SLAM-like iterative refinement loop inside ZING. SLAM handles pointers by doing an apriori alias analysis, and using predicates to refine the imprecision in alias analysis. We believe that directly handling pointers in the abstraction will scale better.

We have used ZING to check stuck-freeness of distributed applications [8, 7], concurrency errors in devicedrivers,⁴ and protocol errors in a replicated file system.⁵ Though a discussion of these applications is beyond the scope of this paper, all of the above algorithms and optimizations were driven by the need to make ZING scale on these applications.

Acknowledgments. We thank Jakob Lichtenberg and Georg Weissenbacher for their efforts in making ZING work inside the SLAM engine. We thank Tom Ball and Byron Cook for several discussions regarding this effort. We thank Vlad Levin for making the ZING UI display error traces in terms of the driver's C code. Abhay Gupta wrote ZING models of a large file-replication protocol. This effort helped uncover several bugs and performance bottlenecks in ZING and one serious bug in the protocol. We thank Tony Hoare and Cedric Fournet for working with us on the theory of stuck-free conformance.

References

- 1. Zing Language Specification http://research.microsoft.com/zing.
- 2. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. Technical report, Microsoft Research, 2004.
- T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In SPIN 00: SPIN Workshop, LNCS 1885, pages 113–130. Springer-Verlag, 2000.
- 4. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, January 2002.
- 5. S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.
- 6. M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *ICSE 01: International Conference on Software Engineering*, pages 177–187. ACM, 2001.
- C. Fournet, C. A. R. Hoare, S. K. Rajamani, and J. Rehof. Stuck-free conformance theory for CCS. Technical Report MSR-TR-2004-09, Microsoft Research, 2004.
- 8. C. Fournet, C.A.R. Hoare, S.K. Rajamani, and J. Rehof. Stuck-free conformance. In *CAV 04: Computer-Aided Verification*, LNCS. Springer-Verlag, 2000.
- 9. C. A. R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985.
- G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineer*ing, 23(5):279–295, May 1997.
- G.J. Holzmann. Logic verification of ANSI-C code with Spin. In SPIN 00: SPIN Workshop, LNCS 1885, pages 131–147. Springer-Verlag, 2000.
- R. Iosif and R. Sisto. dSPIN: A dynamic extension of SPIN. In SPIN 99: SPIN Workshop, LNCS 1680, pages 261–276. Springer-Verlag, 1999.

⁴ Jakob Lichtenberg and Georg Weissenbacher started this work as an intern project in the summer of 2003.

⁵ Abhay Gupta did this work as an intern project in the summer of 2003.

15

- 13. R. J. Lipton. Reduction: A method of proving properties of parallel programs. In *Communications of the ACM*, volume 18:12, pages 717–721, 1975.
- 14. R. Milner. Communicating and Mobile Systems: the π -Calculus. Cambridge University Press, 1999.
- S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *POPL 04: ACM Principles of Programming Languages*, pages 245– 255. ACM, 2004.
- S. K. Rajamani and J. Rehof. Conformance checking for models of asynchronous message passing software. In *CAV 02: Computer-Aided Verification*, LNCS 2404, pages 166–179. Springer-Verlag, 2002.
- 17. G. Ramalingam. Context sensitive synchronization sensitive analysis is undecidable. ACM Trans. on Programming Languages and Systems, 22:416–430, 2000.
- Robby, M. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *FSE 03: Foundations of Software Engineering*, pages 267– 276. ACM, 2003.
- 19. A. W. Roscoe. The Theory and Practice of Concurrency. Prentice Hall, 1998.
- S. D. Stoller. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*, 4(1):71–91, October 2002.
- W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In ICASE 00: Automated Software Engineering, pages 3–12, 2000.

A Semantics for Concurrent Separation Logic

Stephen Brookes

Carnegie Mellon University Department of Computer Science

Abstract. We present a denotational semantics based on action traces, for parallel programs which share mutable data and synchronize using resources and conditional critical regions. We introduce a resource-sensitive logic for partial correctness, adapting separation logic to the concurrent setting, as proposed by O'Hearn. The logic allows program proofs in which "ownership" of a piece of state is deemed to transfer dynamically between processes and resources. We prove soundness of this logic, using a novel "local" interpretation of traces, and we show that every provable program is race-free.

1 Introduction

Parallel programs involve the concurrent execution of processes which share state and are intended to cooperate interactively. It is notoriously difficult to ensure absence of runtime errors such as *races*, in which one process changes a piece of state being used by another process, and *dangling pointers*, which may occur if two processes attempt simultaneously to deallocate the same storage. Such phenomena can cause unpredictable or irreproducible behavior.

Rather than relying on assumptions about the granularity of hardware primitives, it is preferable to use program design rules and proof techniques that guarantee error-freedom. The classic example is the syntax-directed logic for partial correctness properties of (pointer-free) parallel programs introduced by Owicki and Gries [15], building on prior work of Hoare [7]. This approach focusses on critical variables, the identifiers concurrently written by one process and read or written by another. The programmer must partition the critical variables among named resources, and each occurrence of a critical variable must be inside a region naming the relevant resource. Assuming that resource management is implemented by a suitable synchronization primitive, such as semaphores [6, 1], the design rules guarantee mutually exclusive access to critical variables and therefore freedom from races. Each process relies on its environment to ensure that when a resource is available the corresponding resource invariant holds, and guarantees that when the process releases the resource the invariant will hold again (cf. rely/guarantee methodology as in [9]). This use of resource invariants abstracts away from what happens "inside" a critical region and focusses on the places where synchronization occurs.

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 16-34, 2004.

[©] Springer-Verlag Berlin Heidelberg 2004

This method works well for simple (pointer-free) parallel programs, but the task of reasoning about parallel pointer-programs is made more difficult by the potential for aliasing, when distinct expressions may denote the same pointer: static design rules no longer suffice to prevent races involving pointer values. For example, the program [x]:=0||[y]:=1 has a race if x and y are aliases, and this cannot be determined from the syntax of the program.

Peter O'Hearn [11, 12] has proposed an adaptation of the Owicki-Gries rules to handle parallel pointer-programs, incorporating ideas from separation logic [17, 14,8]. The main technical novelty in this adaptation involves the use of *separating conjunction* in the rules dealing with resource invariants and parallel composition. Although this may appear superficially to produce "obvious" variants of the traditional rules, the original rules (using the standard form of conjunction) are unsound for pointer-programs, and soundness of the new rules is far from obvious. Indeed, Reynolds has shown that O'Hearn's rules are unsound without restrictions on resource invariants [18, 13].

O'Hearn provides a series of compelling examples with informal correctness proofs, but (as he remarks) the logic cannot properly be assessed without a suitable semantic model [11]. Such a model is not readily available in the literature: traditional models for concurrency do not include pointers or race-detection, and models for pointer-programs do not typically handle concurrency. In this paper we give a denotational semantics, using *action traces*, that solves these problems, using a form of parallel composition that detects races and treats them as catastrophic¹. Our semantic model embodies a classic principle of concurrent program design, originally articulated by Dijkstra [6] and echoed in the design of the classic inference rules for shared-memory programs [7, 15]:

... processes should be loosely connected; by this we mean that apart from the (rare) moments of explicit intercommunication, the individual processes are to be regarded as completely independent of each other.

In other words, concurrent processes do not interfere (or cooperate) except through explicit synchronization. Our semantics makes this idea concrete through the interplay between traces, which describe interleaved behaviors of processes, and an enabling relation on "local states" that models "no interference from outside except at synchronization". This interplay permits a formalization of O'Hearn's "processes that mind their own business" [12], and leads to a Parallel Decomposition Lemma that reflects the intuition behind Dijkstra's principle in a semantically precise manner.

The Owicki-Gries logic and O'Hearn's adaptation assume a fixed collection of resources and a fixed set of parallel processes. We reformulate O'Hearn's inference rules in a more semantically natural manner, allowing statically scoped resource declarations and nested parallel compositions. We assume that each resource invariant is a *precise* separation logic formula, so that every time a program acquires or releases a resource there is a uniquely determined portion of

¹ This idea was suggested by John Reynolds [18].

the heap whose ownership can be deemed to transfer. We give a suitably general (and compositional) notion of validity, and we prove that the proof rules, using precise invariants, are sound. Our soundness proof demonstrates that a verified program has no race conditions.

We omit proofs, and we do not include examples to illustrate the logic; the reader should see O'Hearn's paper [12] for such examples, which may be replicated quite straightforwardly in our more formal setting. O'Hearn's paper also discusses the limitations of the logic and identifies opportunities for further research. We assume familiarity with the syntax and semantics of separation logic [17]. Apart from this we have tried to include enough technical detail to make the paper self-contained.

2 Syntax

Our programming language combines shared-memory parallelism with pointer operations. The syntax for *commands* (ranged over by c) is given by the following abstract grammar, in which r ranges over *resource names*, i over *identifiers*, e over *integer expressions*, and b over *boolean expressions*:

$$c ::= \mathbf{skip} \mid i:=e \mid i:=[e] \mid [e]:=e' \mid i:=\mathbf{cons} (e_0, \dots, e_n) \mid \mathbf{dispose} \mid e \mid c_1; c_2 \mid c_1 \parallel c_2 \mid \mathbf{if} \mid b \mathbf{then} \mid c_1 \mathbf{else} \mid c_2 \mid \mathbf{while} \mid b \mathbf{do} \mid c \mid \mathbf{resource} \mid r \mathbf{in} \mid c \mid \mathbf{with} \mid r \mathbf{when} \mid b \mathbf{do} \mid c \mid c_1 \mathbf{else} \mid c_2 \mid c_1 \mid c_2 \mid \mathbf{vert} \mid c_1 \mathbf{else} \mid c_2 \mid c_1 \mid c_2 \mid \mathbf{vert} \mid c_1 \mathbf{else} \mid c_2 \mid \mathbf{vert} \mid c_2 \mid \mathbf{vert} \mid c_1 \mathbf{else} \mid c_2 \mid \mathbf{vert} \mid c_1 \mathbf{else} \mid c_1 \mathbf{else} \mid c_2 \mid c_1 \mid c_2 \mid c_1 \mid c_2 \mid c_1 \mid c_1 \mathbf{else} \mid c_1 \mathbf{else} \mid c_2 \mid c_1 \mid c_1 \mathbf{else} \mid c_1 \mathbf{else} \mid c_2 \mid c_1 \mid c_1 \mathbf{else} \mid c_2 \mid c_1 \mid c_2 \mid c_1 \mid c_2 \mid c_1 \mid c_1 \mathbf{else} \mid c_1 \mathbf{else} \mid c_2 \mid c_1 \mid c_2 \mid c_1 \mid c_1 \mathbf{else} \mid c_1 \mathbf{else} \mid c_2 \mid c_1 \mid c_2 \mid c_1 \mid c_2 \mid c_1 \mid c_1 \mathbf{else} \mid c_1 \mathbf{else} \mid c_2 \mid c_1 \mid c_2 \mid c_1 \mid c_2 \mid c_1 \mid c_1 \mathbf{else} \mid c_1 \mathbf{else} \mid c_2 \mid c_1 \mid c_2 \mid c_1 \mid c_2 \mid c_1 \mathbf{else} \mid c_1 \mathbf{else} \mid c_2 \mid c_1 \mid c_1 \mathbf{else} \mid c_1 \mathbf{else} \mid c_2 \mid c_1 \mid c_1 \mid c_2 \mid c_1 \mid c_2 \mid c_1 \mid c_1 \mathbf{else} \mid c_2 \mid c_1 \mid c_1 \mathbf{else} \mid c_2 \mid c_1 \mid c_2 \mid c_2 \mid c_1 \mid c_2 \mid c_2 \mid c_2 \mid c_2 \mid c_2 \mid c_1 \mid c_2 \mid c_$$

Expressions are *pure*, so evaluation has no side-effect and the value of an expression depends only on the *store*. An *assignment* command i:=e affects only the store; *allocation* $i:=cons(e_0, \ldots, e_n)$, *lookup* i:=[e], *update* [e]:=e', and *disposal* **dispose**(e) involve the *heap*. A command of form **resource** r **in** c introduces a local resource name r, whose scope is c. A command of form **with** r when b do c is a *conditional critical region* for resource r. A process attempting to enter a region must wait until the resource is available, acquire the resource and evaluate b: if b is **true** the process executes c then releases the resource; if b is **false** the process releases the resource and waits to try again. A resource can only be held by one process at a time. We use the abbreviation with r do c when b is **true**.

Let free(c) be the set of identifiers occurring free in c, with a similar notation for expressions. Let writes(c) be the set of identifiers having a free write occurrence in c, and res(c) be the set of resource names occurring free in c. These sets are defined as usual, by structural induction. For instance, $res(with r when b do c) = res(c) \cup \{r\}$, $res(resource r in c) = res(c) - \{r\}$.

3 Semantics

We give a trace-theoretic semantics for expressions and commands. The meaning of an expression will be a set of trace-value pairs, and the meaning of a command will be a set of traces. The trace set denoted by a program describes in abstract

19

terms the possible interactive computations that the program may perform when executed fairly, in an environment which is also capable of performing actions. We interpret sequential composition as concatenation of traces, and parallel composition as a resource-sensitive form of interleaving of traces that enforces mutually exclusive access to each resource. By presenting traces as sequences of *actions* we can keep the underlying notion of *state* more or less implicit.² We will exploit this feature later, when we use the semantics to prove soundness of a concurrent separation logic. We start by providing an interpretation of actions using a global notion of state; later we will set up a more refined local notion of state with which it is easier to reason about ownership.

Our semantics is designed to support reasoning about partial correctness and the absence (or potential presence) of runtime errors. The semantics also models deadlock, as a form of infinite waiting, and allows reasoning about safety and liveness properties. The semantics assumes that parallel processes are executed under the control of a *weakly fair* scheduler [16], so that each process that has not yet terminated will eventually be scheduled for execution.

States, Actions, and Traces

A value is either an integer, or an address. We use v to range over values, l over addresses. Let V_{int} be the set of integers, V_{addr} be the set of addresses³, and V_{bool} be the set of truth values. A resource set is a finite set of resource names. A state σ comprises a store s, a heap h, and a resource set A. The store maps a finite set of identifiers to values; the heap maps a finite set of addresses to values. We use notations such as $[i_1 : v_1, \ldots, i_k : v_k]$ and $[l_1 : v_1, \ldots, l_n : v_n]$ to denote stores and heaps, and $[s \mid i : v]$ and $[h \mid l : v]$ for updated stores and heaps. We write $s \setminus X$ for the store obtained by removing the identifiers in X from the domain of s, and $h \setminus l$ for the heap obtained from h by deleting l from its domain. When heaps h_1 and h_2 have disjoint domains we write $h_1 \perp h_2$, and we let $h_1 \cdot h_2$ denote their union. We use a similar notation for stores. An "initial" state will have the form $(s, h, \{\})$; we may use the abbreviation (s, h) in such a case.

We will describe a program's behavior in terms of *actions*. These include *store actions*: reads i=v and writes i:=v to identifiers; *heap actions*: lookups [l]=v, updates [l]:=v, allocations $alloc(l, [v_0, \ldots, v_n])$, and disposals disp(l) of addresses; and *resource actions*: try(r), acq(r), rel(r) involving resource names. We also include an *idle* action δ , and an error action *abort*. We use λ to range over the set of actions.

Each action λ is characterized by its *effect*, a partial function $\stackrel{\lambda}{\Longrightarrow}$ from states to states. This partial function describes the set of states in which the action

² An advantage of action traces [4, 3] over the *transition traces* [16, 5] often used to model shared-memory parallel languages is succinctness: an action typically acts the same way on many states, and we can express this implicitly, without enumerating all pairs of states related by the action.

³ Actually we treat addresses as integers, so that our semantic model can incorporate address arithmetic, but for moral reasons we distinguish between integers as values and integers which happen to be addresses in current use.

is enabled, and the state change caused by executing the action. Note that an action may cause a runtime error, for which we employ the error state **abort**.

Definition 1. The effect $\stackrel{\lambda}{\Rightarrow}$ of an action λ is given by the following clauses:

$(s,h,A) \stackrel{o}{\Longrightarrow} (s,h,A)$	always
$(s,h,A) \xrightarrow{i=v} (s,h,A)$	$if(i,v)\in s$
$(s,h,A) \xrightarrow{i=v} ext{abort}$	$\textit{if } i \not\in \texttt{dom}(s)$
$(s,h,A) \xrightarrow{i:=v} ([s \mid i:v],h,A)$	$\textit{if } i \in \texttt{dom}(s)$
$(s,h,A) \xrightarrow{i:=v} abort$	$\textit{if } i \not\in \texttt{dom}(s)$
$(s,h,A) \xrightarrow{[l]=v} (s,h,A)$	$\textit{if}~(l,v) \in h$
$(s,h,A) \xrightarrow{[l]=v} ext{abort}$	$\textit{if } l \not\in \texttt{dom}(h)$
$(s,h,A) \xrightarrow{[l]:=v} (s,[h \mid l:v],A)$	$\textit{if } l \in \texttt{dom}(h)$
$(s,h,A) \xrightarrow{[l]:=v} ext{abort}$	$\textit{if } l \not\in \texttt{dom}(h)$
$(s,h,A) \xrightarrow{alloc(l,[v_0,\ldots,v_n])} (s,[h \mid l:v_0,\ldots,l+n:v_n],A)$	$if \forall m \leq n. \ l+m \notin dom(h)$
$(s,h,A) \xrightarrow{disp(l)} (s,h \setminus l,A)$	$if \ l \in \mathtt{dom}(h)$
$(s,h,A) \xrightarrow{disp(l)} abort$	$\textit{if } l \not\in \texttt{dom}(h)$
$(s,h,A) \xrightarrow{try(r)} (s,h,A)$	$\textit{if } r \in A$
$(s,h,A) \xrightarrow{acq(r)} (s,h,A \cup \{r\})$	$if r ot \in A$
$(s,h,A) \xrightarrow{rel(r)} (s,h,A-\{r\})$	$if r \in A$
$(s,h,A) \xrightarrow{abort} abort$	always
$abort \stackrel{\lambda}{\Longrightarrow} abort$	always

It is obvious from the above definition that store actions depend only on the store, heap actions depend only on the heap, and resource actions depend only on the resource set. In general an action is either *enabled* or *stuck* in a given state. For example, if s(x) = 0 the action x=0 is enabled, but the action x=1 is stuck. The stuck cases play only a minor role in the development.

Note that a try(r) action is allowed, from a state (s, h, A) in which $r \in A$, to model the case where one parallel component of the program has already acquired r but another component process wants to acquire it and must wait until the resource is released. A process can only acquire a resource that it does not already possess, and can only release a resource that it currently holds.

The clause defining the effect of an allocation action is non-deterministic, to model our assumption that storage management is governed by a mutual exclusion discipline and ensures the use of "fresh" heap cells. A given state (s, h, A) enables all allocation actions of the form $alloc(l, [v_0, \ldots, v_n])$ for which the heap cells $l, l + 1, \ldots, l + n$ are all outside of dom(h). We assume that the storage allocator never chooses to allocate a heap cell in current use, so we do not need to include an error case for allocate actions. On the other hand, since disposals are done by the program we include an error case for disposal actions to account for the possibility of a dangling pointer.

A *trace* is a non-empty finite or infinite sequence of actions. Let **Tr** be the set of all traces. We use α, β as meta-variables ranging over the set of traces, and T_1, T_2 range over trace sets.

We write $\alpha\beta$ for the trace obtained by concatenating α and β ; when α is infinite this is just α . We assume that α abort $\beta = \alpha$ abort, and $\alpha\delta\beta = \alpha\beta$, for all traces α and β .

For trace sets T_1 and T_2 we let $T_1 T_2 = \{\alpha_1 \alpha_2 \mid \alpha_1 \in T_1 \& \alpha_2 \in T_2\}$, and we use the usual notations T^* and T^{ω} for the finite and infinite concatenations of traces from the set T. We let $T^{\infty} = T^* \cup T^{\omega}$.

We define the effect $\stackrel{\alpha}{\Longrightarrow}$ of a trace α in the obvious way, by composing the effects of the actions occurring in the trace. When $(s, h, A) \stackrel{\alpha}{\Longrightarrow} (s', h', A')$ the trace α can be executed from (s, h, A) without the need for interference from outside; we call such a trace *sequential*⁴. As is well known, the sequential traces of $c_1 || c_2$ cannot generally be determined from the sequential traces of c_1 and c_2 , so we need to include non-sequential traces in order to achieve a compositional semantics.

Parallel Composition

The behavior of a command depends on resources: those held by the command and those being used by its environment. These sets of resources start empty and will always be disjoint. Accordingly we define for each action λ a *resource enabling* relation $(A_1, A_2) \xrightarrow{\lambda} (A'_1, A_2)$ on disjoint pairs of resource sets, to specify when a process holding resources A_1 , in an environment that holds A_2 , can perform this action, and the action's effect on the resources held by the program:

$$\begin{array}{l} (A_1, A_2) \xrightarrow{try(r)} (A_1, A_2) \\ (A_1, A_2) \xrightarrow{acq(r)} (A_1 \cup \{r\}, A_2) & \text{if } r \notin A_1 \cup A_2 \\ (A_1, A_2) \xrightarrow{rel(r)} (A_1 - \{r\}, A_2) & \text{if } r \in A_1 \\ (A_1, A_2) \xrightarrow{\lambda} (A_1, A_2) & \text{if } \lambda \text{ is not a resource action} \end{array}$$

Clearly if A_1 and A_2 are disjoint and $(A_1, A_2) \xrightarrow{\lambda} (A'_1, A'_2)$ then $A_2 = A'_2$ and A'_1 is disjoint from A_2 .

This resource enabling notion generalizes in the obvious way to a sequence of actions; we write $(A_1, A_2) \xrightarrow{\alpha} \cdot$ to indicate that a process holding resources A_1 in an environment holding A_2 can perform the trace α .

We want to detect *race conditions* caused by an attempt to write to an identifier or address being used concurrently. This can be expressed succinctly as follows. First, we extend the definitions of free and writes to actions:

⁴ Technically we say that α is sequential if and only if $\stackrel{\alpha}{\Longrightarrow} \neq \{\}$.
Informally, $free(\lambda)$ contains the identifiers or addresses whose current values are needed to enable the action, and $writes(\lambda)$ contains the identifiers or addresses whose values in the current state are affected by the action. We do not include addresses $l, \ldots, l + n$ in the free- or write-set of $alloc(l, [v_0, \ldots, v_n])$, because these addresses are assumed to be fresh when the action occurs.

We write $\lambda_1 \bowtie \lambda_2$ (λ_1 interferes with λ_2) when λ_1 and λ_2 represent a race:

$$\lambda_1 \bowtie \lambda_2 \Leftrightarrow \texttt{free}(\lambda_1) \cap \texttt{writes}(\lambda_2) \neq \{\} \lor \texttt{writes}(\lambda_1) \cap \texttt{free}(\lambda_2) \neq \{\}.$$

Notice that we do not regard two concurrent reads as a disaster.

We then define, for each pair (A_1, A_2) of disjoint resource sets and each pair (α_1, α_2) of action sequences, the set $\alpha_{1A_1}|_{A_2}\alpha_2$ of all *mutex fairmerges* of α_1 using A_1 with α_2 using A_2 . The definition is inductive⁵ in the lengths of α_1 and α_2 , and we include the empty sequence ϵ , to allow a simpler formulation:

$$\begin{aligned} \alpha_{1 A_{1}} \|_{A_{2}} \epsilon &= \{ \alpha_{1} \mid (A_{1}, A_{2}) \xrightarrow{\alpha_{1}} \cdot \} \\ \epsilon_{A_{1}} \|_{A_{2}} \alpha_{2} &= \{ \alpha_{2} \mid (A_{2}, A_{1}) \xrightarrow{\alpha_{2}} \cdot \} \\ (\lambda_{1}\alpha_{1})_{A_{1}} \|_{A_{2}} (\lambda_{2}\alpha_{2}) &= \{ abort \} \quad \text{if } \lambda_{1} \bowtie \lambda_{2} \\ &= \{ \lambda_{1}\beta \mid (A_{1}, A_{2}) \xrightarrow{\lambda_{1}} (A_{1}', A_{2}) \& \beta \in \alpha_{1 A_{1}'} \|_{A_{2}} (\lambda_{2}\alpha_{2}) \} \\ &\cup \{ \lambda_{2}\beta \mid (A_{2}, A_{1}) \xrightarrow{\lambda_{2}} (A_{2}', A_{1}) \& \beta \in (\lambda_{1}\alpha_{1})_{A_{1}} \|_{A_{2}'} \alpha_{2} \} \\ &\quad \text{otherwise} \end{aligned}$$

For traces α_1 and α_2 , let $\alpha_1 \| \alpha_2$ be defined to be $\alpha_{1\{\}} \|_{\{\}} \alpha_2$. For trace sets T_1 and T_2 we define $T_1 \| T_2 = \bigcup \{ \alpha_1 \| \alpha_2 \mid \alpha_1 \in T_1 \& \alpha_2 \in T_2 \}$.

Semantics of Expressions

An expression will denote a set of *evaluation traces paired with values*: we define $\llbracket e \rrbracket \subseteq \mathbf{Tr} \times V_{int}$ for an integer expression e, and $\llbracket b \rrbracket \subseteq \mathbf{Tr} \times V_{bool}$ for a boolean expression b. Since expression values depend only on the store, the only non-trivial actions participating in such traces will be reads. To allow for the possibility of interference during expression evaluation we include both non-sequential and sequential evaluation traces. Again the sequential traces describe what happens if an expression is evaluated without interference.

The semantic functions are given, by structural induction, in the usual way. For example:

⁵ We can also give a *coinductive* definition of the mutex fairmerges of two infinite traces, starting from a given disjoint pair of resource sets. We need mostly to work here with finite traces, so we omit the details.

$$\begin{split} & \llbracket 10 \rrbracket = \{ (\delta, 10) \} \\ & \llbracket i \rrbracket = \{ (i = v, v) \mid v \in V_{int} \} \\ & \llbracket e_1 + e_2 \rrbracket = \{ (\rho_1 \rho_2, v_1 + v_2) \mid (\rho_1, v_1) \in \llbracket e_1 \rrbracket \And (\rho_2, v_2) \in \llbracket e_2 \rrbracket \} \\ & \llbracket (e_0, \dots, e_n) \rrbracket = \{ (\rho_0 \dots \rho_n, [v_0, \dots, v_n]) \mid \forall j. \ 0 \le j \le n \Rightarrow (\rho_j, v_j) \in \llbracket e_j \rrbracket \}. \end{split}$$

The use of concatenation in these semantic clauses assumes that sum expressions and lists are evaluated in left-right order. This assumption is not crucial; it would be just as reasonable to assume parallel evaluation for such expressions. With an appropriately modified semantic definition, this adjustment can be made without affecting the ensuing development.

Let $\llbracket b \rrbracket_{\mathbf{true}} \subseteq \mathbf{Tr}$ be the set of traces ρ such that $(\rho, \mathbf{true}) \in \llbracket b \rrbracket$, and $\llbracket b \rrbracket_{\mathbf{false}}$ be the set of traces ρ such that $(\rho, \mathbf{false}) \in \llbracket b \rrbracket$.

Semantics of Commands

A command *c* denotes a trace set $[c] \subseteq \mathbf{Tr}$, defined by structural induction.

Definition 2.

The trace set **[***c***]** *of a command c is defined by the following clauses:*

$$\begin{split} & [\mathbf{skip}] = \{\delta\} \\ & [i:=e] = \{\rho \, i:=v \mid (\rho, v) \in [\![e]] \} \\ & [i:=[e]] = \{\rho \, [v]=v' \, i:=v' \mid (\rho, v) \in [\![e]] \} \\ & [i:=\operatorname{cons}\,(e_0, \dots, e_n)] = \{\rho \, alloc(l, L) \, i:=l \mid (\rho, L) \in [\![(e_0, \dots, e_n)]\!] \} \\ & [[e]:=e'] = \{\rho \, \rho' \, [v]:=v' \mid (\rho, v) \in [\![e]] \& (\rho', v') \in [\![e']] \} \\ & [dispose(e)] = \{\rho \, disp(l) \mid (\rho, l) \in [\![e]] \} \\ & [c_1; c_2] = [\![c_1]] \, [\![c_2]] = \{\alpha_1 \alpha_2 \mid \alpha_1 \in [\![c_1]] \& \alpha_2 \in [\![c_2]] \} \\ & [if \ b \ then \ c_1 \ else \ c_2] = [\![b]\!]_{true} \, [\![c_1]] \cup [\![b]\!]_{false} \, [\![c_2]] \\ & [while \ b \ do \ c] = ([\![b]\!]_{true} \, [\![c]])^* \, [\![b]\!]_{false} \ \cup ([\![b]\!]_{true} \, [\![c]])^{\omega} \\ & [\![c_1]|c_2] = [\![c_1]] | [\![c_2]] \\ & [whith \ r \ when \ b \ do \ c] = wait^* \ enter \ \cup \ wait^{\omega} \\ & where \ wait = acq(r) \, [\![b]\!]_{false} \ rel(r) \ \cup \ \{try(r)\} \\ & and \quad enter = acq(r) \, [\![b]\!]_{true} \, [\![c]\!] \ rel(r) \\ & [resource \ r \ in \ c] = \{\alpha \setminus r \mid \alpha \in [\![c]]_r\} \end{split}$$

In the above semantic clauses we have prescribed a left-to-right sequential evaluation order for $i:=cons(e_0, \ldots, e_n)$ and [e]:=e', reflected in the use of concatenation on the traces of sub-expressions; again this assumption is not crucial, and it is straightforward to adapt the ensuing development to allow for parallel evaluation of sub-expressions.

The iterative structure of the traces of a conditional critical region reflect its use to achieve synchronization: waiting until the resource is available and the test condition is true, followed by execution of the body command while holding the resource, and finally releasing the resource. Note the possibility that the body may loop forever or encounter a runtime error, in which case the resource will not get released. Since $[true]_{false} = \{\}$ and $[true]_{true} = \{\delta\}$ it is easy to derive a simpler formula for the trace set of with r do c: we have

$$\llbracket \mathbf{with} \ r \ \mathbf{do} \ c \rrbracket = try(r)^* \ acq(r) \llbracket c \rrbracket \ rel(r) \ \cup \ \{try(r)^{\omega}\}.$$

TEAM LING

Since the command **resource** r in c introduces a local resource named r, whose scope is c, its traces are obtained from traces of c in which r is assumed initially available and the actions involving r are executed without interference. We let $[c]_r$ be the set of traces of c which are *sequential for* r in this manner⁶. We let $\alpha \setminus r$ be the trace obtained from α by replacing each action on r by δ .

Examples

- 1. $[x:=x+1] = \{x=v : x:=v+1 \mid v \in V_{int}\}$ This program always terminates, when executed from a state in which x has a value; its effect is to increment the value of x by 1.
- 2. $[x:=x+1||x:=x+1] = \{x=v \text{ abort } | v \in V_{int}\}$ Concurrent assignments to the same identifier cause a race, no matter what the initial value of x is.
- 3. **[with** r do x:=x+1] = $try(r)^* acq(r)$ [x:=x+1] $rel(r) \cup \{try(r)^{\omega}\}$ This program needs to acquire r before incrementing x, and will wait forever if the resource never becomes available.
- 4. **[with** r do x:=x + 1 **[with** r do x:=x + 1]] contains traces of the forms $acq(r) \alpha rel(r) acq(r) \beta rel(r)$, $acq(r) \alpha rel(r) try(r)^{\omega}$, and $try(r)^{\omega}$, where $\alpha, \beta \in [x:=x+1]$, as well as traces of similar form containing additional try(r) steps. Only the first kind are sequential for r. It follows that

 $[[resource r in (with r do x:=x+1]||with r do x:=x+1)]] = [\alpha\beta | \alpha, \beta \in [[x:=x+1]] = [[x:=x+1]; x:=x+1]].$

The overall effect is the same as that of two consecutive increments. 5. The command x:=cons(1)||y:=cons(2) has the trace set

$$\{alloc(l, [1]) x := l \mid l \in V_{addr}\} \| \{alloc(l', [2]) y := l' \mid l' \in V_{addr} \}.$$

This set includes traces of the form

$$alloc(l, [1]) x := l alloc(l, [2]) y := l,$$

and other interleavings of alloc(l, [1]) x := l with alloc(l, [2]) y := l, none of which are sequential. The set also includes traces obtained by interleaving alloc(l, [1]) x := l and alloc(l', [2]) y := l', where $l \neq l'$; all of these are sequential.

6. The command dispose(x) || dispose(y) has trace set

$$\{x=v \operatorname{disp}(v) \mid v \in V_{\operatorname{addr}}\} \|\{y=v' \operatorname{disp}(v') \mid v' \in V_{\operatorname{addr}}\}.$$

⁶ Technically, we say that α is *sequential for* r if $(\{\}, \{\}, \{\}) \xrightarrow{\alpha[r]} \cdot$ holds, where $\alpha[r]$ is the subsequence of α consisting of actions on resource r. This expresses formally the requirement that α represents an execution in which r is initially available and r is never acquired (or released) by the environment. Equivalently, $\alpha[r]$ is a prefix of a trace in the set $(acq(r) try(r)^{\infty} rel(r))^{\infty}$. Note in particular that $try(r)^{\omega}$ is not sequential for r.

25

This includes traces of the form x=v y=v abort because of the race-detecting clause in the definition of fairmerge. If this command is executed from a state in which x and y are aliases a race will occur, with both processes attempting to dispose the same heap cell: if s(x) = s(y) = v and $v \in dom(h)$ we have $(s, h, \{\}) \xrightarrow{x=v y=v abort} abort$.

4 Concurrent Separation Logic

Separation logic [17] provides a class of formulas for specifying properties of stores and heaps. The syntax includes separating conjunction, denoted $p_1 * p_2$, and formulas **emp** and $e \mapsto e'$ specifying an empty heap and a singleton heap. We write $(s, h) \models p$ when (s, h) satisfies p. In particular, $(s, h) \models p_1 * p_2$ if and only if there are disjoint heaps h_1, h_2 such that $h = h_1 \cdot h_2$, $(s, h_1) \models p_1$, and $(s, h_2) \models p_2$. Reynolds [17] provides a Hoare-style partial correctness logic for sequential pointer-programs in which the pre- and post-conditions are separation logic formulas.

We now introduce resource-sensitive partial correctness formulas of the form $\Gamma \vdash \{p\}c\{q\}$, where p and q are separation logic formulas, c is a parallel pointerprogram, and Γ is a *resource context* $r_1(X_1) : R_1, \ldots, r_k(X_k) : R_k$ associating resource names r_j with protection lists X_j and resource invariants R_j . Each protection list represents a finite set of identifiers. We require each resource invariant to be a *precise* separation logic formula. A separation logic formula pis *precise* [17] if for all s and h, there is at most one $h' \subseteq h$ such that $(s, h') \models p$.

Let $\operatorname{dom}(\Gamma) = \{r_1, \ldots, r_k\}$ be the set of resource names in Γ , $\operatorname{owned}(\Gamma) = \bigcup_{j=1}^k X_j$ be the set of identifiers protected by Γ , and $\operatorname{free}(\Gamma) = \bigcup_{j=1}^k \operatorname{free}(R_j)$ be the set of identifiers mentioned in the invariants. Let $\operatorname{inv}(\Gamma) = R_1 * \cdots * R_k$ be the separating conjunction of the resource invariants in Γ . In particular, when Γ is empty this is **emp.** Since each resource invariant is precise it follows that $\operatorname{inv}(\Gamma)$ is precise.

We will impose some syntactic *well-formedness* constraints on contexts and formulas, designed to facilitate modularity. Specifically:

- Γ is well-formed if its entries are disjoint, in that if $i \neq j$ then $r_i \neq r_j$, $X_i \cap X_j = \{\}$, and $\mathbf{free}(R_i) \cap X_j = \{\}$. - $\Gamma \vdash \{p\}c\{q\}$ is well-formed if Γ is well-formed, and p and q do not mention
- $-\Gamma \vdash \{p\}c\{q\}$ is well-formed if Γ is well-formed, and p and q do not mention any protected identifiers, i.e. $free(p,q) \cap owned(\Gamma) = \{\}$.

Thus in a well-formed context each identifier belongs to at most one resource. We do *not* require that the free identifiers in a resource invariant be protected, i.e. that $free(R_i) \subseteq X_i$. This allows us to use a resource invariant to connect the values of protected identifiers and the values of non-critical variables.

The inference rules will enforce the following syntactic constraints on commands, relative to the relevant resource context⁷:

⁷ We will not formalize these properties or give a proof that they hold in all provable formulas. We state them explicitly since they recall analogous requirements in the Owicki-Gries logic and in O'Hearn's rules.

- Every critical identifier is protected by a resource.
- Every free occurrence of a protected identifier is within a region for the corresponding resource.
- Every free write occurrence of an identifier mentioned in a resource invariant is within a region for the corresponding resource.

Intuitively, a resource-sensitive partial correctness formula specifies how a program behaves when executed in an environment that respects the resource context, assuming that at all times the *separating conjunction* of the resource invariants holds, for the currently available resources. The program guarantees to stay within these bounds, provided it can rely on its environment to do likewise. This informal notion of validity for formulas should help provide intuition for the structure of the following inference rules. Later we will give a formal definition of validity.

We allow all well-formed instances of the following inference rules. Some of the rules have side conditions to ensure well-formedness and the syntactic requirements given above, as in [12].

```
- Skip
```

$$\overline{\Gamma \vdash \{p\}\mathbf{skip}\{p\}}$$

- ASSIGNMENT

$$\overline{\varGamma \vdash \{[e/i]p\}i{:=}e\{p\}}$$

$$\text{if } i \not\in \texttt{owned}(\Gamma) \cup \texttt{free}(\Gamma)$$

– Lookup

$$\overline{\Gamma \vdash \{[e'/i]p \land e \mapsto e'\}i:=[e]\{p \land e \mapsto e'\}}$$

if $i \notin \texttt{free}(e, e')$ and $i \notin \texttt{owned}(\Gamma) \cup \texttt{free}(\Gamma)$

- ALLOCATION

 $\overline{\Gamma \vdash \{\operatorname{emp}\} i := \operatorname{cons}(e_0, \dots, e_n) \{i \mapsto e_0 * \dots * i + n \mapsto e_n\}}$ if $i \notin \operatorname{free}(e_0, \dots, e_n)$ and $i \notin \operatorname{owned}(\Gamma) \cup \operatorname{free}(\Gamma)$

- Update

$$\overline{\Gamma \vdash \{e \mapsto -\}[e] := e'\{e \mapsto e'\}}$$

- DISPOSAL

$$\overline{\varGamma \vdash \{e \mapsto -\} \text{dispose } e\{\text{emp}\}}$$

SEQUENTIAL

$$\frac{\Gamma \vdash \{p_1\}c_1\{p_2\} \quad \Gamma \vdash \{p_2\}c_2\{p_3\}}{\Gamma \vdash \{p_1\}c_1; c_2\{p_3\}}$$

- CONDITIONAL

$$\frac{\Gamma \vdash \{p \land b\}c_1\{q\} \quad \Gamma \vdash \{p \land \neg b\}c_2\{q\}}{\Gamma \vdash \{p\} \text{if } b \text{ then } c_1 \text{ else } c_2\{q\}}$$

– Loop

$$\frac{\Gamma \vdash \{p \land b\}c\{p\}}{\Gamma \vdash \{p\} \text{while } b \text{ do } c\{p \land \neg b\}}$$

- PARALLEL $\frac{\Gamma \vdash \{p_1\}c_1\{q_1\} \quad \Gamma \vdash \{p_2\}c_2\{q_2\}}{\Gamma \vdash \{p_1 * p_2\}c_1 \| c_2\{q_1 * q_2\}}$ if free $(p_1, q_1) \cap$ writes $(c_2) =$ free $(p_2, q_2) \cap$ writes $(c_1) = \{\}$ and (free $(c_1) \cap$ writes $(c_2)) \cup$ (free $(c_2) \cap$ writes $(c_1)) \subseteq$ owned (Γ)
- RESOURCE

$$\frac{\Gamma, r(X) : R \vdash \{p\}c\{q\}}{\Gamma \vdash \{p * R\} \text{resource } r \text{ in } c\{q * R\}}$$

- RENAMING RESOURCE

$$\frac{\Gamma \vdash \{p\} \text{resource } r' \text{ in } [r'/r]c\{q\}}{\Gamma \vdash \{p\} \text{resource } r \text{ in } c\{q\}}$$

if $r' \not\in \mathbf{res}(c)$

- REGION

$$\frac{\Gamma \vdash \{(p * R) \land b\}c\{q * R\}}{\Gamma, r(X) : R \vdash \{p\} \text{with } r \text{ when } b \text{ do } c\{q\}}$$

- FRAME

$$\frac{\Gamma \vdash \{p\}c\{q\}}{\Gamma \vdash \{p * I\}c\{q * I\}}$$

$$\text{if } \texttt{free}(I) \cap \texttt{writes}(c) = \{\}$$

- CONSEQUENCE

$$\frac{p' \Rightarrow p \quad \Gamma \vdash \{p\}c\{q\} \quad q \Rightarrow q'}{\Gamma \vdash \{p'\}c\{q'\}}$$

provided $p' \Rightarrow p$ and $q \Rightarrow q'$ are universally valid

- AUXILIARY

$$\frac{\Gamma \vdash \{p\}c\{q\}}{\Gamma \vdash \{p\}c \backslash X\{q\}}$$

if X is auxiliary for c, and $X \cap free(p,q) = \{\}$.

- CONJUNCTION

$$\frac{\Gamma \vdash \{p_1\}c\{q_1\}}{\Gamma \vdash \{p_1 \land p_2\}c\{q_1 \land q_2\}} \frac{\Gamma \vdash \{p_2\}c\{q_2\}}{\Gamma \vdash \{p_1 \land p_2\}c\{q_1 \land q_2\}}$$

- EXPANSION

$$\frac{\Gamma \vdash \{p\}c\{q\}}{\Gamma, \Gamma' \vdash \{p\}c\{q\}}$$

if writes(c) \cap free(Γ') = {} and free(c) \cap owned(Γ') = {}

- CONTRACTION

$$\frac{\varGamma, \varGamma' \vdash \{p\}c\{q\}}{\varGamma \vdash \{p\}c\{q\}}$$

if $\operatorname{res}(c) \subseteq \operatorname{dom}(\Gamma)$

Comments

- 1. The rules dealing with the sequential program constructs are natural adaptations of the rules given by Reynolds [17], with the incorporation of a resource context and side conditions to ensure well-formedness and adherence to the protection policy. The FRAME and CONSEQUENCE rules similarly generalize analogous rules from the sequential setting.
- 2. The PARALLEL, REGION and RESOURCE rules are based on O'Hearn's adaptations of Owicki-Gries inference rules. A side condition in the PARALLEL rule enforces the requirement that each critical variable must be associated with a resource, just as in the original Owicki-Gries rule.
- 3. The AUXILIARY rule similarly adapts the Owicki/Gries rule for auxiliary variables⁸. As usual, a set of identifiers X is said to be *auxiliary* for c if every free occurrence in c of an identifier from X is in an assignment that only affects the values of identifiers in X. In particular, auxiliary identifiers cannot occur in conditional tests or loop tests, and do not influence the control flow of the program. The command $c \setminus X$ is obtained from c by deleting assignments to identifiers in X.
- 4. In the RESOURCE RENAMING rule we write [r'/r]c for the command obtained from c by replacing each free occurrence of r by r'.
- 5. We have omitted the obvious structural rules permitting permutation of resource contexts.

5 Validity

We wish to establish that every provable resource-sensitive formula is *valid*, but we need to determine precisely what that should mean. Adapting the notion of validity familiar from the sequential setting, we might try to interpret validity of $\Gamma \vdash \{p\}c\{q\}$ as the property that every finite computation of c from a state satisfying $p * inv(\Gamma)$ is error-free and ends in a state satisfying $q * inv(\Gamma)$. However, this notion of "sequential validity" is not compositional for parallel programs; although it expresses a desirable property we need a notion of validity that takes account of process interaction.

Informally we might say that the formula $\Gamma \vdash \{p\}c\{q\}$ is valid if every finite interactive computation of c from a state satisfying $p*inv(\Gamma)$, in an environment that respects Γ , is error-free, also respects Γ , and ends in a state satisfying

⁸ Owicki and Gries cite Brinch Hansen [2] and Lauer [10] as having first recognized the need for auxiliary variables in proving correctness properties of concurrent programs.

 $q * inv(\Gamma)$. However, such a formulation would be incomplete, since it does not properly specify what "respect" for Γ entails. To obtain a suitably formal (and compositional) notion of validity we need to keep track of the portions of the state deemed to be "owned" by a process, its environment, and the available resources.

With respect to a resource context Γ , a process holding resource set A should be allowed to access identifiers protected by resources in A, but not identifiers protected by other resources. We say that (s, h, A) is a *local state* consistent with Γ if dom $(s) \cap$ owned $(\Gamma) =$ owned $(\Gamma \mid A)$, where $\Gamma \mid A$ is the subset of Γ involving resources in A. We let $\Gamma \setminus A$ be the rest of Γ . We introduce *local enabling relations* between local states: a step $(s, h, A) \xrightarrow{\lambda}_{\Gamma'} (s', h', A')$ means that in state (s, h, A)a process can perform action λ , causing its local state to change to (s', h', A')and respecting the resource invariants and protection rules. We use the error state **abort** to handle runtime errors and logical errors such as an attempt to release a resource in a state for which no sub-heap satisfies the corresponding invariant, or a write to an identifier mentioned in a resource invariant without first acquiring the resource.

Definition 3. The local enabling relations $\frac{\lambda}{\Gamma}$ are the least relations satisfying the following clauses, in which (s, h, A) ranges over local states consistent with the relevant context:

$(s,h,A) \xrightarrow{\delta}{\Gamma} (s,h,A)$	always
$(s,h,A) \xrightarrow[\Gamma]{abort} abort$	always
$(s,h,A) \xrightarrow{i=v}{\Gamma} (s,h,A)$	$if~(i,v)\in s$
$(s,h,A) \xrightarrow{i=v}{\Gamma} ext{abort}$	$\textit{if } i \not\in \texttt{dom}(s)$
$(s,h,A) \xrightarrow{i:=v} \Gamma ([s \mid i:v],h,A)$	$\textit{if } i \in \texttt{dom}(s) - \texttt{free}(\Gamma \backslash A)$
$(s,h,A) \xrightarrow{i:=v}{\Gamma}$ abort	$if \ i ot\in \texttt{free}(\Gamma ackslash A)$
$(s,h,A) \xrightarrow{[l]=v}{\Gamma} (s,h,A)$	$\textit{if}~(l,v) \in h$
$(s,h,A) \xrightarrow{[l]=v}{\Gamma} \text{abort}$	$\textit{if } l \not\in \texttt{dom}(h)$
$(s,h,A) \xrightarrow{[l]:=v'}{\Gamma} (s,[h \mid l:v'],A)$	$\textit{if } l \in \texttt{dom}(h)$
$(s,h,A) \xrightarrow{[l]:=v'} \mathbf{abort}$	$\textit{if } l \not\in \texttt{dom}(h)$
$(s,h,A) \xrightarrow{alloc(l,[v_0,\ldots,v_n])}_{\Gamma} (s,[h \mid l:v_0,\ldots,l+n:v_n],A)$	$\textit{if } \forall m \leq n. \ l+m \not\in dom(h)$
$(s,h,A) \xrightarrow{disp(l)} (s,h \backslash l,A)$	$\textit{if } l \in \texttt{dom}(h)$
$(s,h,A) \xrightarrow{disp(l)} \operatorname{abort}$	$\textit{if } l \not\in \texttt{dom}(h)$
$(s,h,A) \xrightarrow{try(r)} (s,h,A)$	always
$(s,h,A) \xrightarrow{acq(r)} (s \cdot s', h \cdot h', A \cup \{r\})$	$if \ r \not\in A, \ h \perp h',$
	$\mathtt{dom}(s') = X, (s\cdot s',h') \models R$
$(s,h,A) \xrightarrow{rel(r)}{\Gamma,r(X):R^*} (s \setminus X, h-h', A-\{r\})$	$\textit{if } r \in A, h' \subseteq h \& (s,h') \models R$
$(s,h,A) \xrightarrow{rel(r)}{\Gamma,r(X):R}$ abort	$\textit{if} \forall h' \subseteq h. \ \neg(s,h') \models R$

The clauses for acq(r) and rel(r) deal with ownership transfer: when a process acquires a resource its local state grows to include the identifiers protected by the resource and the heap portion in which the resource invariant holds; when a process releases a resource its local state ceases to include the protected identifiers and the heap associated with the resource invariant; a "logical" error occurs if the invariant is not suitably satisfied. Since resource invariants are assumed to be precise formulas in each case there is a uniquely determined portion of heap associated with the relevant invariant.

We write $\sigma \xrightarrow{\alpha}{\Gamma} \sigma'$ when there is a local computation α from σ to σ' .

Note that non-sequential traces play a non-trivial role in the local enabling relation, and in a local computation external interference can occur only at a resource acquisition step. Thus the local enabling relation provides a formalization of "loosely connected" processes in the spirit of Dijkstra.

The following result connects the local enabling relations $\frac{\alpha}{\Gamma}$, which model interactive execution in an environment that respects a resource context, and the effect relations $\stackrel{\alpha}{\Longrightarrow}$, which represent interference-free executions, when α is a sequential trace.

Lemma 1 (Empty Transfer Lemma)

Let α be a finite trace, let $\{r_1, \ldots, r_n\}$ be the set of resource names occurring in actions of α , and let Γ_0 be the resource context $r_1(\{\}) : \operatorname{emp}, \ldots, r_n(\{\}) : \operatorname{emp}$. Then $(s, h, A) \xrightarrow{\alpha} \sigma'$ if and only if $(s, h, A) \xrightarrow{\alpha}{\Gamma_0} \sigma'$.

Theorem 2 (Respect for Resources)

If $\alpha \in [c]$ and $(s, h, A) \xrightarrow{\alpha} (s', h', A')$, then $\operatorname{dom}(s') = \operatorname{dom}(s)$ and A = A'.

Note that these results imply the corresponding property for sequential traces.

Corollary 3

If $\alpha \in \llbracket c \rrbracket$ and $(s, h, A) \stackrel{\alpha}{\Longrightarrow} (s', h', A')$, then $\operatorname{dom}(s) = \operatorname{dom}(s')$ and A = A'.

The following *parallel decomposition* property relates a local computation of a parallel program to local computations of its components. If the critical identifiers of c_1 and c_2 are protected by resources in Γ , a local computation of $c_1 \| c_2$ can be "projected" into a local computation of c_1 and a local computation of c_2 . In stating this property we use (s, h) as an abbreviation for $(s, h, \{\})$.

Theorem 4 (Parallel Decomposition)

Suppose $(\operatorname{free}(c_1) \cap \operatorname{writes}(c_2)) \cup (\operatorname{writes}(c_1) \cap \operatorname{free}(c_2)) \subseteq \operatorname{owned}(\Gamma)$ and $\alpha \in \alpha_1 || \alpha_2$, where $\alpha_1 \in [\![c_1]\!]$ and $\alpha_2 \in [\![c_2]\!]$. Suppose $h_1 \perp h_2$ and $h = h_1 \cdot h_2$.

- $If(s,h) \xrightarrow{\alpha}{\Gamma} abort then$ $(s \setminus writes(c_2), h_1) \xrightarrow{\alpha_1}{\Gamma} abort or (s \setminus writes(c_1), h_2) \xrightarrow{\alpha_2}{\Gamma} abort.$ - $If(s,h) \xrightarrow{\alpha}{\Gamma} (s',h') then$ $(s \setminus writes(c_2), h_1) \xrightarrow{\alpha_1}{\Gamma} abort or (s \setminus writes(c_1), h_2) \xrightarrow{\alpha_2}{\Gamma} abort,$ or there are disjoint heaps $h'_1 \perp h'_2$ such that $h' = h'_1 \cdot h'_2$ and • $(s \setminus writes(c_2), h_1) \xrightarrow{\alpha_1}{\Gamma} (s' \setminus writes(c_2), h'_1)$ • $(s \setminus writes(c_1), h_2) \xrightarrow{\alpha_2}{\Gamma} (s' \setminus writes(c_1), h'_2)$ The definition of local enabling formalizes the notion of a computation by a process, in an environment that respects resources, and "minds its own business" by obeying the ownership policy of a given resource context. This leads us to the following rigorous formulation of validity. Again we write (s, h) for $(s, h, \{\})$.

Definition 5.

The formula $\Gamma \vdash \{p\}c\{q\}$ is valid if for all traces α of c, all local states (s,h) such that dom $(s) \supseteq \texttt{free}(c, \Gamma) - \texttt{owned}(\Gamma)$, and all σ' , if $(s,h) \models p$ and $(s,h) \xrightarrow{\alpha}{\Gamma} \sigma'$ then $\sigma' \neq \texttt{abort}$ and $\sigma' \models q$.

This definition uses the local enabling relation, so that the quantification ranges over local states (s, h) consistent with Γ , for which $dom(s) \cap owned(\Gamma) = \{\}$. Furthermore, this notion of validity involves *all* traces of *c*, not just the sequential traces and not just the finite traces⁹.

When Γ is the empty context and $\operatorname{res}(c) = \{\}$, validity of $\{\} \vdash \{p\}c\{q\}$ implies the usual notion of partial correctness together with the guaranteed absence of runtime errors. More generally, the same implication holds when $\operatorname{res}(c) = \{r_1, \ldots, r_n\}$ and Γ is the context $r_1(\{\}) : \operatorname{emp}, \ldots, r_n(\{\}) : \operatorname{emp}$.

We now come to the main result of this paper: soundness of our logic.

Theorem 6 (Soundness)

Every provable formula $\Gamma \vdash \{p\}c\{q\}$ is valid.

Proof:

Show that each well formed instance of an inference rule is sound: if the rule's premisses and conclusion are well formed, the side conditions hold, and the premisses are valid, then the conclusion is valid. It then follows, by induction on the length of the derivation, that every provable formula is valid.

We give details for the PARALLEL rule.

- PARALLEL COMPOSITION

Suppose that $\Gamma \vdash \{p_1\}c_1\{q_1\}$ and $\Gamma \vdash \{p_2\}c_2\{q_2\}$ are well formed and valid, and that $\texttt{free}(p_1,q_1) \cap \texttt{writes}(c_2) = \texttt{free}(p_2,q_2) \cap \texttt{writes}(c_1) = \{\}$ and $(\texttt{free}(c_1) \cap \texttt{writes}(c_2)) \cup (\texttt{writes}(c_1) \cap \texttt{free}(c_2)) \subseteq \texttt{owned}(\Gamma).$

It is clear that $\Gamma \vdash \{p_1 * p_2\}c_1 \| c_2\{q_1 * q_2\}$ is well formed. We must show that $\Gamma \vdash \{p_1 * p_2\}c_1 \| c_2\{q_1 * q_2\}$ is valid.

Let $(s,h) \models p_1 * p_2$, and suppose $h_1 \perp h_2$, $h = h_1 \cdot h_2$, and $(s,h_1) \models p_1$, $(s,h_2) \models p_2$. Since $\texttt{free}(p_1) \cap \texttt{writes}(c_2) = \texttt{free}(p_2) \cap \texttt{writes}(c_1) = \{\}$ we also have $(s \setminus \texttt{writes}(c_2), h_1) \models p_1$ and $(s \setminus \texttt{writes}(c_1), h_2) \models p_2$.

Let $\alpha \in [c_1 || c_2]$, and $(s, h) \xrightarrow{\alpha}{\Gamma} \sigma'$. Choose traces $\alpha_1 \in [c_1]$ and $\alpha_2 \in [c_2]$ such that $\alpha \in \alpha_1 || \alpha_2$. If $\sigma' = \text{abort}$ the Parallel Decomposition Lemma would imply that $(s \setminus \texttt{writes}(c_2), h_1) \xrightarrow{\alpha_1}{\Gamma} \texttt{abort}$ or $(s \setminus \texttt{writes}(c_1), h_2) \xrightarrow{\alpha_2}{\Gamma} \texttt{abort}$. Neither of these is possible, since they contradict the assumed validity of

⁹ The infinite traces only really matter in the no-**abort** requirement, since we never get $\sigma \xrightarrow{\alpha}{\Gamma} \sigma'$ when α is infinite and σ' is a proper state.

the premisses $\Gamma \vdash \{p_1\}c_1\{q_1\}$ and $\Gamma \vdash \{p_2\}c_2\{q_2\}$. If α is infinite that is all we need. Otherwise α is finite, and σ' has the form (s', h'). Again by the Parallel Decomposition Lemma and validity of the premisses, there are heaps $h'_1 \perp h'_2$ such that $h' = h'_1 \cdot h'_2$,

$$\begin{array}{l} (s \setminus \texttt{writes}(c_2), h_1) \xrightarrow[\Gamma]{} \alpha_1 \\ \hline \\ (s \setminus \texttt{writes}(c_1), h_2) \xrightarrow[\Gamma]{} \alpha_2 \\ \hline \\ (s' \setminus \texttt{writes}(c_1), h_2) \xrightarrow[\Gamma]{} (s' \setminus \texttt{writes}(c_1), h_2'), \end{array}$$

and $(s' \setminus writes(c_2), h'_1) \models q_1, (s' \setminus writes(c_1), h'_2) \models q_2$. Since q_1 does not depend on writes (c_2) and q_2 does not depend on writes (c_1) we also have $(s', h'_1) \models q_1$ and $(s', h'_2) \models q_2$, from which it follows that $(s', h') \models q_1 * q_2$, as required.

6 **Provability Implies No Races**

For a process holding resource set A and a corresponding global state (s, h, A), let $s \downarrow A = s \setminus owned(\Gamma \setminus A)$. This is the "local" portion of the global store "visible" to the process by virtue of its current resource set.

The following result shows how the local effect of an action relates to its global effect, modulo the protection policy imposed by the resource context, assuming that the process performing the action owns resources in A and the global heap contains a sub-heap in which the resource invariants for the available resources hold, separately.

Lemma 7 (Connection Property)

Let (s, h, A) be a global state and suppose $h = h_1 \cdot h_2$ with $(s, h_2) \models inv(\Gamma \setminus A)$.

- $If(s, h, A) \xrightarrow{\lambda} abort then(s \downarrow A, h_1, A) \xrightarrow{\lambda} abort.$
- $If(s,h,A) \stackrel{\lambda}{\Longrightarrow} (s',h',A')$ then

 - either (s↓A, h₁, A) ^λ/_Γ abort
 or there are heaps h'₁ ⊥ h'₂ such that h' = h'₁ ⋅ h'₂, (s', h'₂) ⊨ inv(Γ\A'), and $(s \downarrow A, h_1, A) \xrightarrow{\lambda} (s' \downarrow A', h'_1, A')$

We can then deduce the following result for all commands c, letting A = A' ={} and using induction on trace structure.

Corollary 8 Let $\alpha \in [c]$. suppose $h = h_1 \cdot h_2$, and $(s, h_2) \models inv(\Gamma)$. - If $(s,h) \stackrel{\alpha}{\Longrightarrow}$ abort then $(s \setminus \text{owned}(\Gamma), h_1) \stackrel{\alpha}{\longrightarrow}$ abort. $- If(s,h) \stackrel{\alpha}{\Longrightarrow} (s',h') then$ • either $(s \setminus owned(\Gamma), h_1) \xrightarrow{\alpha}{\Gamma} abort,$ • or there are heaps $h'_1 \perp h'_2$ such that $h' = h'_1 \cdot h'_2$, $(s', h'_2) \models inv(\Gamma)$, and $(s \setminus owned(\Gamma), h_1) \xrightarrow{\alpha}{\Gamma} (s' \setminus owned(\Gamma), h'_1)$.

33

Finally, combining this with the definition of validity we obtain a link with the earlier notion of "sequential validity", which we can express rigorously in terms of the interference-free enabling relations $\stackrel{\alpha}{\Longrightarrow}$.

Theorem 9 (Valid Implies Race-Free)

If $\Gamma \vdash \{p\}c\{q\}$ is valid and well formed, then c is error-free from every global state satisfying $p * inv(\Gamma)$. More specifically, for all states σ, σ' and all traces $\alpha \in [c]$, if $\sigma \models p * inv(\Gamma)$ and $\sigma \stackrel{\alpha}{\Longrightarrow} \sigma'$ then $\sigma' \neq abort$ and $\sigma' \models q * inv(\Gamma)$.

Combining this result with the Soundness Theorem, it follows that provability of $\Gamma \vdash \{p\}c\{q\}$ implies that c is race-free from all states satisfying $p * inv(\Gamma)$.

Acknowledgements

I have benefitted immensely from discussions with Peter O'Hearn, whose ideas from [11] prompted this work; John Reynolds, who suggested treating races catastrophically; and Josh Berdine, whose insights led to technical improvements. The anonymous referees also made helpful suggestions.

References

- 1. P. Brinch Hansen. *Structured multiprogramming*. Comm. ACM, 15(7):574-578, July 1972.
- 2. P. Brinch Hansen. *Concurrent programming concepts*. ACM Computing Surveys 5(4):223-245, December 1973.
- S. Brookes, *Traces, pomsets, fairness and full abstraction for communicating processes.* Proc. CONCUR 2002, Brno. Springer LNCS vol. 2421, pp. 466-482. August 2002.
- S. Brookes. Communicating Parallel Processes: Deconstructing CSP. In: Millenium Perspectives in Computer Science, Proc. 1999 Oxford-Microsoft Symposium in honour of Sir Tony Hoare. Palgrave, 2000.
- 5. S. Brookes. *Full abstraction for a shared-variable parallel language*. Inf. Comp., vol 127(2):145-163, Academic Press, June 1996.
- E. W. Dijkstra. *Cooperating sequential processes*. In: Programming Languages, F. Genuys (editor), pp. 43-112. Academic Press, 1968.
- C.A.R. Hoare, *Towards a Theory of Parallel Programming*. In Operating Systems Techniques, C. A. R. Hoare and R. H. Perrott, editors, pp. 61-71, Academic Press, 1972.
- S. Isthiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. Proc. 28th POPL conference, pp. 36-49, January 2001.
- 9. C.B. Jones. Specification and design of (parallel) programs. Proc. IFIP Conference, 1983.
- 10. H.C. Lauer. *Correctness in operating systems*. Ph. D. thesis, Carnegie Mellon University, 1973.
- 11. P. W. O'Hearn. Notes on separation logic for shared-variable concurrency. Unpublished manuscript, January 2002.

- 12. P.W. O'Hearn. *Resources, Concurrency, and Local Reasoning.* This volume, Springer LNCS, CONCUR 2004, London, August 2004.
- P.W. O'Hearn, H. Yang, and J.C. Reynolds. Separation and Information Hiding. Proc. 31st POPL conference, pp 268-280, Venice. ACM Press, January 2004.
- 14. P. W. O'Hearn and D. J. Pym. *The logic of bunched implications*. Bulletin of Symbolic Logic, 5(2):215-244, June 1999.
- 15. S. Owicki and D. Gries, Verifying properties of parallel programs: An axiomatic approach, Comm. ACM. 19(5):279-285, May 1976.
- D. Park, On the semantics of fair parallelism. In: Abstract Software Specifications, Springer-Verlag LNCS vol. 86, 504–526, 1979.
- J.C. Reynolds, Separation logic: a logic for shared mutable data structures, Invited paper. Proc. 17th IEEE Conference on Logic in Computer Science, LICS 2002, pp. 55-74. IEEE Computer Society, 2002.
- J. C. Reynolds. Lecture notes on separation logic (15-819A3), chapter 8, page 178. Department of Computer Science, Carnegie-Mellon University, Spring 2003. Revised May 23, 2003.

A Survey of Regular Model Checking

Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Mayank Saksena

Dept. of Information Technology, P.O. Box 337, S-751 05 Uppsala, Sweden {parosh, bengt, marcusn, mayanks}@it.uu.se

Abstract. Regular model checking is being developed for algorithmic verification of several classes of infinite-state systems whose configurations can be modeled as words over a finite alphabet. Examples include parameterized systems consisting of an arbitrary number of homogeneous finite-state processes connected in a linear or ring-formed topology, and systems that operate on queues, stacks, integers, and other linear data structures. The main idea is to use regular languages as the representation of sets of configurations, and finite-state transducers to describe transition relations. In general, the verification problems considered are all undecidable, so the work has consisted in developing semi-algorithms, and decidability results for restricted cases. This paper provides a survey of the work that has been performed so far, and some of its applications.

1 Introduction

A significant research effort is currently being devoted to extending the applicability of algorithmic verification to parameterized and infinite-state systems, using approaches based on abstraction, deductive techniques, decision procedures, etc. One major approach is to extend the paradigm of symbolic model checking [BCMD92] to new classes of models by an appropriate symbolic representation; examples include timed automata, systems with unbounded communication channels, Petri nets, and systems that operate on integers and reals.

Regular model checking is such an extension, in which sets of states and transition relations are represented by regular sets, typically over finite or infinite words or tree structures. Most work has considered models whose configurations can be represented as finite words of arbitrary length over a finite alphabet. This includes parameterized systems consisting of an arbitrary number of homogeneous finite-state processes connected in a linear or ring-formed topology, and systems that operate on queues, stacks, integers, and other linear data structures. Regular model checking was advocated by Kesten et al. [KMM+01] and by Boigelot and Wolper [WB98], as a uniform framework for analyzing several classes of parameterized and infinite-state systems. The idea is that regular sets will provide an efficient representation of infinite state spaces, and play a role similar to that played by Binary Decision Diagrams (BDDs) for symbolic model checking of finite-state systems. One can also exploit automata-theoretic algorithms for manipulating regular sets. Such algorithms have been successfully implemented, e.g., in the Mona [HJJ+96] system.

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 35-48, 2004.

[©] Springer-Verlag Berlin Heidelberg 2004

A generic task in symbolic model checking is to compute properties of the set of reachable states, in order to verify safety properties. For finite-state systems this is typically done by state-space exploration, but for infinite-state systems this procedure terminates only if there is a bound on the distance (in number of transitions) from the initial configurations to any reachable configuration. An analogous observation holds if we perform a reachability analysis backwards, by iteration-based methods [CES86,QS82] from a set of "unsafe" configurations. A parameterized or infinite-state system does not have such a bound, and any nontrivial model checking problem is undecidable. In contrast to deductive application of systems like Mona [BK98], the goal in regular model checking is to verify system properties algorithmically. An important challenge is therefore to devise so-called acceleration techniques, which calculate the effect of arbitrarily long sequences of transitions. This problem has been addressed in regular model checking [JN00,BJNT00,AJNd02]. In general, the effect of acceleration is not computable. However, computability have been obtained for certain classes [JN00]. Analogous techniques for computing accelerations have successfully been developed for several classes of parameterized and infinite-state systems, e.g., systems with unbounded FIFO channels [BG96,BGWW97,BH97,ABJ98], systems with stacks [BEM97,Cau92,FWW97,ES01], and systems with counters [BW94,CJ98].

In this paper, we survey the available work on regular model-checking. The use of regular sets to model and specify systems is discussed in Section 2. Techniques for computing invariants and reachable loops are surveyed in Section 3. Finally, some extensions are discussed in Section 4.

2 Framework

Model checking is concerned with automated analysis of transition systems, each consisting of

- a set of configurations (or states), some of which are initial, and
- a transition relation, which is a binary relation on the set of configurations.

The configurations represent possible "snapshots" of the system state, and the transition relation describes how these can evolve over time. Most work on model checking assumes that the set of configurations is finite, but significant effort is underway to develop model checking techniques for transition systems with infinite sets of configurations.

In its simplest form, the regular model checking framework represents a transition system as follows.

- A configuration (state) of the system is a word over a finite alphabet Σ .
- The set of *initial configurations* is a regular set over Σ .
- The transition relation is a regular and length-preserving relation on Σ^* . It is represented by a finite-state transducer over $(\Sigma \times \Sigma)$, which accepts all words $(a_1, a'_1) \cdots (a_n, a'_n)$ such that $(a_1 \cdots a_n, a'_1 \cdots a'_n)$ is in the transition relation. Sometimes, the transition relation is given as a union of a finite number of relations, each of which is called an *action*.

Given a transducer T, we often abuse notation and use T also to denote the relation defined by the transducer. For a set S of configurations and a binary relation T on configurations, let $S \circ T$ denote the set of configurations w such that w' T w for some $w' \in S$, let T^+ denote the transitive closure of T and T^* denote the reflexive transitive closure of T. Let S^2 denote the set of pairs of elements in S.

In the regular model checking framework it is possible to model parameterized systems with linear or ring-shaped topologies, e.g., by letting each position in the word model the state of a system component. It is also possible to model programs that operate on linear unbounded data structures such as queues, stacks, integers, etc. For instance, a stack can be modeled by letting each position in the word represent a position in the stack. The restriction to length-preserving transducers implies that we cannot dynamically "create" new stack positions. Therefore the stack should initially contain an arbitrary but bounded number of empty stack positions, which are "statically allocated". We can then faithfully model all finite computations of the system, by initially allocating sufficiently many empty stack positions. However, it may not be possible to model faithfully all infinite computations of the system. Thus, the restriction to length-preserving transducers introduces no limitations for analyzing safety properties, but may incur restrictions on the ability to specify and verify liveness properties of systems with dynamically allocated data structures.

2.1 Examples

In Figure 1 we consider a token passing protocol: a simple parameterized system consisting of an arbitrary (but finite) number of processes organized in a linear fashion. Initially, the left-most process has the token. In each step, the process currently having the token passes it to the right. A configuration of the system is a word over the alphabet $\{t, n\}$, where t represents that the process has the token, and n represents not having it. For instance, the word *nntnn* represents a configuration of a system with five processes where the third process has the token. The set of initial states is given by the regular expression tn^* (Figure 1(a)). The transition relation is represented by the transducer in Figure 1(b). For instance, the transducer accepts the word (n,n)(n,n)(t,n)(n,t)(n,n), representing the pair (nntnn, nnntn) of configurations where the token is passed from the third to the fourth process.



Fig. 1. Initial set of states and transition relation

As a second example, we consider a system consisting of a finite-state process operating on one unbounded FIFO channel. Let Q be the set of control states of the process, and let M be the (finite) set of messages which can reside inside the channel. A configuration of the system is a word over the alphabet $Q \cup M \cup \{e\}$, where the *padding symbol* e represents an empty position in the channel. For instance the word $q_1em_3m_1ee$ corresponds to a configuration where the process is in state q_1 and the channel (of length four) contains the messages m_3 and m_1 in this order. The set of configurations of the system can thus be described by the regular expression $Qe^*M^*e^*$.

By allowing arbitrarily many padding symbols e, one can model channels of arbitrary but bounded length. As an example, the action where the process sends the message m to the channel and changes state from q_1 to q_2 is modeled by the transducer in Figure 2. In the figure, "M" is used to denote any message in M.



Fig. 2. Transducer for sending a message *m* to the channel

2.2 Verification Problems

We will consider two types of verification problems in this paper.

The first problem is verification of *safety properties*. A safety property is of form "bad things do not happen during system execution". A safety property can be verified by solving a *reachability* problem. Formulated in the regular model checking framework, the corresponding problem is the following: given a set of initial configurations I, a regular set of *bad configurations* B and a transition relation specified by a transducer T, does there exist a path from I to B through the transition relation T? This amounts to checking whether $(I \circ T^*) \cap B = \emptyset$. The problem can be solved by computing the set $Inv = I \circ T^*$ and checking whether it intersects B.

The second problem is verification of *liveness properties*. A liveness property is of form "a good thing happens during system execution". Often, liveness properties are verified using fairness requirements on the model, which can state that certain actions must infinitely often be either disabled or executed. Since, by the restriction to length-preserving transducers, any infinite system execution can only visit a finite set of configurations, the verification of a liveness property can be reduced to a *repeated reachability* problem. The repeated reachability problem

39

asks, given a set of initial configurations I, a set of *accepting configurations* F and a transition relation T, whether there exists an infinite computation from I through T that visits F infinitely often. By letting F be the configurations where the fairness requirement is satisfied, and by excluding states where the "good thing" happens from T, the liveness property is satisfied if and only if the repeated reachability problem is answered negatively.

Since the transition relation is length-preserving, and hence each execution can visit only a finite set of configurations, the repeated reachability problem can be solved by checking whether there exists a reachable loop containing some configuration from F. This can be checked by computing $(Inv \cap F)^2 \cap Id$ and checking whether this relation intersects T^+ . Here Id is the identity relation on the set of configurations, and $Inv = I \circ T^*$ is as before.

Sets like $I \circ T^*$ and relations like T^+ are in general not regular or even computable (note that T could model the computation steps of a Turing machine). Even if they are regular, they are sometimes not effectively computable. In these cases, the above verification problems cannot be solved by the proposed techniques. Therefore, a main challenge in regular model checking is to design semi-algorithms which successfully compute such sets and relations for as many examples as possible. In Section 3, we briefly survey some techniques that have been developed for this purpose.

2.3 A Specification Logic

The translation from a problem of verifying liveness under fairness requirements to a repeated reachability problem can be rather tricky. One way to make the task easier is to provide an intuitive syntax for modeling and specification, which can be automatically translated to repeated reachability problems, in analogy with the way that linear-time temporal logic formulas are translated to Büchi automata [VW86].

A logic LTL(MSO) was proposed for regular model checking in $[AJN^+04]$. It uses a MSO (monadic second-order logic) over finite words to specify regular sets, and LTL to specify temporal properties. The problem of model checking a formula in LTL(MSO) can be automatically translated into a repeated reachability problem $[AJN^+04]$.

The logic LTL(MSO) combines (under certain restrictions) temporal operators of LTL [KPR98], including \Box (always) and \diamond (eventually), and MSO quantification over positions (first-order) and sets of positions (second-order). Models of LTL(MSO) formulas are sequences of configurations (i.e., words), where the first-order position variables denote positions in configurations, and the secondorder variables denote sets of positions. For instance, if $\varphi(i)$ is a formula which specifies a temporal property at position *i* in the word, then the formula $\forall i \diamond \varphi(i)$ specifies that $\varphi(i)$ eventually holds at each position in the word.

In LTL(MSO), one can represent the configuration of a system by *configuration predicates*, which can be seen as Boolean arrays indexed by positions. For instance, in the token passing example, we can introduce a configuration predicate t, where the atomic formula t[i] is interpreted as "the process at position i has the token", and t'[i] as "the process at position *i* will have the token in the next time step".

Example. Our running example, token passing, is modeled in LTL(MSO) below following the style of TLA [Lam94], where the system and the property of interest are both specified by formulas. The local states of processes are represented by a configuration predicate t - for every i, we have that t[i] is true if and only if process i has the token. The set of initial states is modeled by **initial**, where only the first process has the token. The transition relation where the token is passed from position i to position i+1 is modeled by **pass**(i). Finally, the entire system model is specified by **system.** The system actions are "one process passes the token, or all processes idle". Models of this formula correspond to runs of the system.

An example of a safety property for this system is "two different processes may not have the token at the same time":

safety =
$$\Box \neg \exists i, j \ (i \neq j \land t[i] \land t[j])$$

In order to specify termination ("the last process eventually gets the token") we add a fairness constraint for the token passing action. For an action α , let enabled(α) represent the set of states where the action α can be taken. enabled(α) can be expressed in the logic, using an existential quantification of the primed configuration predicates in α .

$$\begin{array}{ll} \mathbf{fairness} &= \forall i \ \Box \diamondsuit (\mathbf{pass}(i) \lor \neg \mathbf{enabled}(\mathbf{pass}(i))) \\ \mathbf{termination} &= \diamondsuit \exists i \ (t[i] \ \land \ \forall j \ \neg (j=i+1)) \end{array}$$

To check that the algorithm satisfies the safety property, we translate the property system $\land \neg$ safety to a reachability problem. To check that the algorithm satisfies the liveness property, we translate the property system \land fairness $\land \neg$ itermination to a repeated reachability problem.

3 Algorithms

In Section 2, we stated a verification problem as that of computing a representation of $I \circ T^*$ (or T^+) for some transition relation T and some set of configurations I. In some cases we also have a set of *bad* configurations B and we want to check whether $I \circ T^* \cap B \neq \emptyset$. Algorithms for symbolic model checking are often based on starting from I and repeatedly applying T. As a running illustration, we will consider the problem of computing the transitive closure T^+ for the transducer in Figure 1(b). A first attempt is to compute T^n , the composition of T with itself n times for $n = 1, 2, 3, \cdots$. For example, T^3 is the transition relation where the token gets passed three positions to the right. Its transducer is given below.



A transducer for T^+ is one where the token gets passed an arbitrary number of times, given below.



The challenge is to derive the above transducer algorithmically. Obviously, it cannot be done naively by simply computing the approximations T^n for $n = 1, 2, 3, \dots$, since this will not converge. Some acceleration or widening techniques must be developed that compute a representation of T^+ by other means. In this section, we present some techniques developed in the literature for that purpose.

3.1 Quotienting

Several techniques in the literature are based on suitable *quotienting* of transducers that represent approximations of T^n for some value(s) of n. This involves finding an *equivalence relation* \simeq on the states of approximations, and to merge equivalent states, obtaining a quotient transducer. For instance, in the transducer that represents T^3 above, we can define the states 1,2, and 3 to be equivalent. By merging them, we obtain the transducer T^3/\simeq which in this example happens to be equivalent to T^+ .

One problem is that quotienting in general increases the language accepted by a transducer: $\mathcal{L}(T^n) \subseteq \mathcal{L}(T^n/\simeq)$, usually with strict inclusion. This problem was resolved in [AJNd02,BJNT00,DLS01,AJMd02] by characterizing equivalence relations \simeq such that T^+ is equivalent to $(T/\simeq)^+$ for any transducer T, i.e., the quotienting does not increase the transitive closure of the transducer. To explain the idea, let us first build explicitly a transducer for T^+ as the union of transducers T^n for $n = 1, 2, 3, \cdots$. Each state of T^n is labeled with a sequence of states from T, resulting from the product construction using n copies of T. The result is called the *history transducer*. The history transducer corresponding to Figure 1(b) is shown below.



Recall minimization algorithms for automata. They are based on building a *forward* bisimulation \simeq_F on the states, and then carry out minimization by quotienting. For instance, in the above history transducer, all states with names of form $2^i 1$ for any $i \ge 0$ are forward bisimilar. Analogously, we can find a backward bisimulation \simeq_B . For instance, all states with names of form 10^i , $i \ge 0$, are backward bisimilar. Dams et al. [DLS01] showed how to combine a forward \simeq_F and a backward bisimulation \simeq_B into an equivalence relation \simeq which preserves the transitive closure of the transducer. In [AJNd03], this result was generalized to consider *simulations* instead of bisimulations. The simulations can be obtained by computing properties of the original automaton T (as in [AJNd02,AJNd03]), or on successive approximations of T^n (as in [DLS01]).

From the results in [AJNd03] it follows for the above history transducer that the states with names in $2^{i}1$ can be merged for $i \ge 1$, and the same holds for 10^{i} . The equivalence classes for that transducer would be 2^{+} , 0^{+} , 10^{+} , $2^{+}1$ and $2^{+}10^{+}$. Hence, it can be quotiented to the following transducer, which can be minimized to the three-state representation shown earlier.



3.2 Abstraction

In recent work, Bouajjani et al. [BHV04] apply *abstraction* techniques to automata that arise in the iterative computation of $I \circ T^*$. When computing the

sequence $I, I \circ T, I \circ T^2, I \circ T^3, \cdots$ the automata that arise in the computation may all be different or may be very large and contain information that is not relevant for checking whether $I \circ T^*$ has a nonempty intersection with the set of bad configurations *B*. Therefore, each iterate $I \circ T^n$ is abstracted by quotienting under some equivalence relation \simeq . In contrast to the techniques of [AJNd02,BJNT00,DLS01,AJMd02], the abstraction does not need to preserve the language accepted, i.e., $(I \circ T^n) / \simeq$ can be any over-approximation of $I \circ T^n$ or even of $I \circ T^*$. The procedure calculates the sequence of approximations of form $(((I \circ T) / \simeq) \circ T) / \simeq \cdots$. Convergence to a limit T^{lim} can be ensured by choosing \simeq to have finite index.

If now $T^{lim} \cap B = \emptyset$, we can conclude (by $\mathcal{L}((I \circ T^*)) \subseteq \mathcal{L}(T^{lim})$) that $I \circ T^*$ has an empty intersection with *B*. Otherwise, we try to trace back the computation from *B* to *I*. If this succeeds, a counterexample has been found, otherwise the abstraction must be refined by using a finer equivalence relation, from which a more exact approximation T^{lim} can be calculated, etc.

The technique relies on defining suitable equivalence relations. One way is to use the automaton for *B*. We illustrate this on the token passing example. Suppose that *B* is given by the automaton in Fig 3(a), denoting that the last process has the token. Each state *q* in an automaton *A* has a *post language* $\mathcal{L}(A, q)$ which is the set of words accepted starting from that state. For example, in the automaton for *B* we have $\mathcal{L}(B,0) = n^*t$ and $\mathcal{L}(B,1) = \{\epsilon\}$. The post languages are used to define \simeq , such that $q \simeq q'$ holds if for all states *r* of *B* we have $\mathcal{L}(A,q) \cap \mathcal{L}(B,r) = \emptyset$ exactly when $\mathcal{L}(A,q') \cap \mathcal{L}(B,r) = \emptyset$. Each equivalence class of \simeq can be represented by a Boolean vector indexed by states of *B*, which is true on position *s* exactly when the equivalence class members have nonempty intersection with $\mathcal{L}(B,s)$. This is one way to get a finite index equivalence relation.

We show an example of an automaton A in Fig 3(b) with its corresponding abstract version in Fig 3(c). Considering the states of A, we observe that the post languages of states 0 and 1 both have a nonempty intersection with the post language n^*t and an empty intersection with the post language containing the empty string. The post language of state 2 have an empty intersection with



(a) Automaton for B



(b) An automaton A



(c) The abstract version of A

Fig. 3. Applying abstraction

 $n^{*}t$ and an nonempty intersection with the post language containing the empty string.

If a *spurious* counterexample is found, i.e. a counterexample occurring when quotienting with an equivalence \simeq , but not in the original system, we need to refine the equivalence and start again. Automata representing parts of the counterexample can be used, in the same way as the automaton *B* above, to define an equivalence. In [BHV04], the equivalence is refined by using *both B* and automata representing parts of the counterexample. This prevents the same counterexample from occurring twice. Using abstraction can potentially greatly reduce the execution time, since we only need to verify that we cannot reach *B* and therefore it may be that less information about the structure of $I \circ T^*$ needs to be stored.

3.3 Extrapolation

Another technique for calculating $I \circ T^*$ is to speed up the iterative computation by *extrapolation* techniques that try to guess the limit. The idea is to detect a repeating pattern – a regular growth – in the iterations, from which one guesses the effect of arbitrarily many iterations. The guess may be exactly the limit, or an approximation of it.

In [BJNT00,Tou01], the extrapolation is formulated in terms of rules for guessing $I \circ T^*$ from observed growth patterns among the approximations $I, I \circ T, I \circ T^2, \cdots$. Following Bouajjani et al. [BJNT00], if I is a regular expression ρ which is a concatenation of form $\rho = \rho_1 \cdot \rho_2$, and in the successive approximations we observe a growth of form $(\rho_1 \cdot \rho_2) \circ T = \rho_1 \cdot \Lambda \cdot \rho_2$ for some regular expression Λ , then the guess for the limit $\rho \circ T^*$ is $\rho_1 \cdot \Lambda^* \cdot \rho_2$. Touili [Tou01] extends this approach to more general situations. One of these is when ρ is a concatenation of form $\rho_1 \cdot \ldots \cdot \rho_n$ and

$$(\rho_1 \cdot \ldots \cdot \rho_n) \circ T = \bigcup_{i=1}^{n-1} \rho_1 \cdot \ldots \cdot \rho_i \cdot \Lambda_i \cdot \rho_{i+1} \cdot \ldots \cdot \rho_n$$

The guess for the limit $\rho \circ T^*$ is in this case

$$\rho_1 \cdot \Lambda_1^* \cdot \rho_2 \cdot \Lambda_2^* \cdot \ldots \cdot \Lambda_{n-1}^* \cdot \rho_n$$

For example, if $\rho = a^*ba^*$ and *T* is a relation which changes an *a* to a *c*, then $\rho \circ T$ is $a^*ca^*ba^* \cup a^*ba^*ca^*$ (i.e., each step adds either ca^* to the left of *b* or a^*c to the right). The above rule guesses the limit $\rho \circ T^*$ to be $a^*(ca^*)^*b(a^*c)^*a^*$. Touili also suggests other, more general, rules.

Having formed a guess ρ' for the limit, we apply a *convergence test* which checks whether $\rho' = (\rho' \circ T) \cup \rho$. If it succeeds, we can conclude that $\rho \circ T^* \subseteq \rho'$. The work in [BJNT00] and [Tou01] also provide results which state that under some additional conditions, we can in fact conclude that $\rho \circ T^* = \rho'$, i.e., that ρ' is the exact limit.

Boigelot et al. [BLW03] extend the above techniques by considering growth patterns for subsequences of $I, I \circ T, I \circ T^2, \cdots$, consisting of infinite sequences

(b) Automaton for $\rho_I \circ T$

of *sample points*, noting that the union of the approximations in any such subsequence is equal to the union of the approximations in the full sequence. They apply this idea to iterate a special case of relations, *arithmetic transducers*, which operate on binary encodings of integers, and give a sufficient criterion for exact extrapolation.

We illustrate these approaches, using our token passing example. From the initial set $\rho_I = tn^*$, we get $\rho_I \circ T = ntn^*$, $\rho_I \circ T^2 = nntn^*$, $\rho_I \circ T^3 = nnntn^*$, and so on. The methods above detect the growth $\rho_I \circ T = n \cdot \rho_I$, and guess that the limit is n^*tn^* . In this case, the completeness results of [BJNT00,Tou01] allow to conclude that the guessed limit is exact.



(a) Automaton for ρ_I



(c) Extrapolated automaton

Fig. 4. Extrapolating token passing

4 Further Directions

In previous sections, we have presented main techniques in regular model checking for the case where system configurations are modeled as finite words, and transition relations are modeled as length-preserving transducers. In this section, we briefly mention some work where these restrictions are lifted.

Non-Length-Preserving Transducers. Lifting the restriction of length-preservation from transducers allows to model more easily dynamic data structures and parameterized systems of processes with dynamic process creation. The techniques have been extended, see, e.g., [DLS01,BLW03].

Infinite Words. The natural extension to modeling systems by infinite words has been considered by Boigelot et al. [BLW04], having the application to real arithmetic in mind. Regular sets and transducers must then be represented by Büchi automata. To avoid the high complexity of some operations on Büchi automata, the approach is restricted to sets that can be defined by weak deterministic Büchi automata.

Finite Trees. Regular sets of trees can in principle be analyzed in the same way as regular sets of words, as was observed also in [KMM⁺01]. With some complications, similar techniques can be used for symbolic verification [AJMd02,BT02]. Some techniques have been implemented and used to verify simple token-passing algorithms [AJMd02], or to perform data-flow analysis for parallel programs with procedures [BT02].

46 P.A. Abdulla et al.

Context Free Languages. Fisman and Pnueli [FP01] use representations of context-free languages to verify parameterized algorithms, whose symbolic verification require computation of invariants that are non-regular sets of finite words. The motivating example is the Peterson algorithm for mutual exclusion among n processes [PS81].

References

- [ABJ98] Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-fly analysis of systems with unbounded, lossy fifo channels. In Proc. 10th Int. Conf, on Computer Aided Verification, volume 1427 of Lecture Notes in Computer Science, pages 305–318, 1998.
- [AJMd02] Parosh Aziz Abdulla, Bengt Jonsson, Pritha Mahata, and Julien d'Orso. Regular tree model checking. In Proc. 14th Int. Conf. on Computer Aided Verification, volume 2404 of Lecture Notes in Computer Science, 2002.
- [AJN⁺04] P.A. Abdulla, B. Jonsson, Marcus Nilsson, Julien d'Orso, and M. Saksena. Regular model checking for MSO + LTL. In Proc. 16th Int. Conf. on Computer Aided Verification, 2004. to appear.
- [AJNd02] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Regular model checking made simple and efficient. In Proc. CONCUR 2002, 13th Int. Conf. on Concurrency Theory, volume 2421 of Lecture Notes in Computer Science, pages 116–130, 2002.
- [AJNd03] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Algorithmic improvements in regular model checking. In Proc. 15th Int. Conf. on Computer Aided Verification, volume 2725 of Lecture Notes in Computer Science, pages 236–248, 2003.
- [BCMD92] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Symbolic model checking: 10²⁰ states and beyond. *Information and Computation*, 98:142– 170, 1992.
- [BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model Checking. In Proc. Intern. Conf. on Concurrency Theory (CONCUR'97). LNCS 1243, 1997.
- [BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In Alur and Henzinger, editors, Proc. 8th Int. Conf. on Computer Aided Verification, volume 1102 of Lecture Notes in Computer Science, pages 1–12. Springer Verlag, 1996.
- [BGWW97] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In Proc. of the Fourth International Static Analysis Symposium, Lecture Notes in Computer Science. Springer Verlag, 1997.
- [BH97] A. Bouajjani and P. Habermehl. Symbolic reachability analysis of fifochannel systems with nonregular sets of configurations. In Proc. ICALP '97, 24th International Colloquium on Automata, Languages, and Programming, volume 1256 of Lecture Notes in Computer Science, 1997.
- [BHV04] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Proc.* 16th Int. Conf. on Computer Aided Verification, 2004. to appear.

- [BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In Emerson and Sistla, editors, Proc. 12th Int. Conf. on Computer Aided Verification, volume 1855 of Lecture Notes in Computer Science, pages 403–418. Springer Verlag, 2000.
- [BK98] D.A. Basin and N. Klarlund. Automata based symbolic reasoning in hardware verification. *Formal Methods in Systems Design*, 13(3):255– 288, November 1998.
- [BLW03] Bernard Boigelot, Axel Legay, and Pierre Wolper. Iterating transducers in the large. In Proc. 15th Int. Conf. on Computer Aided Verification, volume 2725 of Lecture Notes in Computer Science, pages 223–235, 2003.
- [BLW04] Bernard Boigelot, Axel Legay, and Pierre Wolper. Omega regular model checking. In K. Jensen and A. Podelski, editors, Proc. TACAS '04, 10th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, volume 2988 of Lecture Notes in Computer Science, pages 561–575. Springer Verlag, 2004.
- [BT02] Ahmed Bouajjani and Tayssir Touili. Extrapolating Tree Transformations. In Proc. 14th Int. Conf. on Computer Aided Verification, volume 2404 of Lecture Notes in Computer Science, 2002.
- [BW94] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In Proc. 6th Int. Conf. on Computer Aided Verification, volume 818 of Lecture Notes in Computer Science, pages 55–67. Springer Verlag, 1994.
- [Cau92] Didier Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106(1):61–86, Nov. 1992.
- [CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. ACM Trans. on Programming Languages and Systems, 8(2):244–263, April 1986.
- [CJ98] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In *CAV'98*. LNCS 1427, 1998.
- [DLS01] D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, 2001.
- [ES01] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In Berry, Comon, and Finkel, editors, Proc. 13th Int. Conf. on Computer Aided Verification, volume 2102 of Lecture Notes in Computer Science, pages 324–336, 2001.
- [FP01] Dana Fisman and Amir Pnueli. Beyond regular model checking. In Proc. 21th Conference on the Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science, December 2001.
- [FWW97] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems (extended abstract). In Proc. Infinity '97, Electronic Notes in Theoretical Computer Science, Bologna, Aug. 1997.
- [HJJ⁺96] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In Proc. TACAS '95, 1th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, volume 1019 of Lecture Notes in Computer Science, 1996.

48 P.A. Abdulla et al.

- [JN00] Bengt Jonsson and Marcus Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In S. Graf and M. Schwartzbach, editors, Proc. TACAS '00, 6th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, volume 1785 of Lecture Notes in Computer Science, 2000.
- [KMM⁺01] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256:93–112, 2001.
- [KPR98] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In Proc. ICALP '98, 25th International Colloquium on Automata, Languages, and Programming, volume 1443 of Lecture Notes in Computer Science, pages 1–16. Springer Verlag, 1998.
- [Lam94] L. Lamport. The temporal logic of actions. *ACM Trans. on Programming Languages and Systems*, 16(3):872–923, May 1994.
- [PS81] G.E. Peterson and M.E. Stickel. Myths about the mutal exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.
- [QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In 5th International Symposium on Programming, Turin, volume 137 of Lecture Notes in Computer Science, pages 337–352. Springer Verlag, 1982.
- [Tou01] T. Touili. Regular Model Checking using Widening Techniques. *Electronic Notes in Theoretical Computer Science*, 50(4), 2001. Proc. Workshop on Verification of Parametrized Systems (VEPAS'01), Crete, July, 2001.
- [VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In Proc. LICS '86, 1st IEEE Int. Symp. on Logic in Computer Science, pages 332–344, June 1986.
- [WB98] Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. 10th Int. Conf. on Computer Aided Verification,* volume 1427 of *Lecture Notes in Computer Science,* pages 88–97, Vancouver, July 1998. Springer Verlag.

Resources, Concurrency and Local Reasoning

Peter W. O'Hearn

Queen Mary, University of London

Abstract. In this paper we show how a resource-oriented logic, separation logic, can be used to reason about the usage of resources in concurrent programs.

1 Introduction

Resource has always been a central concern in concurrent programming. Often, a number of processes share access to system resources such as memory, processor time, or network bandwidth, and correct resource usage is essential for the overall working of a system. In the 1960s and 1970s Dijkstra, Hoare and Brinch Hansen attacked the problem of resource control in their basic works on concurrent programming [8,9,11,12,1,2]. In addition to the use of synchronization mechanisms to provide protection from inconsistent use, they stressed the importance of *resource separation* as a means of controlling the complexity of process interactions and reducing the possibility of time-dependent errors. This paper revisits their ideas using the formalism of separation logic [22].

Our initial motivation was actually rather simple-minded. Separation logic extends Hoare's logic to programs that manipulate data structures with embedded pointers. The main primitive of the logic is its separating conjunction, which allows local reasoning about the mutation of one portion of state, in a way that automatically guarantees that other portions of the system's state remain unaffected [16]. Thus far separation logic has been applied to sequential code but, because of the way it breaks state into chunks, it seemed as if the formalism might be well suited to shared-variable concurrency, where one would like to assign different portions of state to different processes.

Another motivation for this work comes from the perspective of general resource-oriented logics such as linear logic [10] and BI [17]. Given the development of these logics it might seem natural to try to apply them to the problem of reasoning about resources in concurrent programs. This paper is one attempt to do so – separation logic's assertion language is an instance of BI – but it is certainly not a final story. Several directions for further work will be discussed at the end of the paper.

There are a number of approaches to reasoning about imperative concurrent programs (e.g., [19,21,14]), but the ideas in an early paper of Hoare on concurrency, "Towards a Theory of Parallel Programming [11]" (henceforth, TTPP), fit particularly well with the viewpoint of separation logic. The approach there revolves around a concept of "spatial separation" as a way to organize thinking about concurrent processes, and to simplify reasoning. Based on compiler-

[©] Springer-Verlag Berlin Heidelberg 2004

enforceable syntactic constraints for ensuring separation, Hoare described formal partial-correctness proof rules for shared-variable concurrency that were beautifully modular: one could reason locally about a process, and simple syntactic checks ensured that no other process could tamper with its state in a way that invalidated the local reasoning.

So, the initial step in this work was just to insert the separating conjunction in appropriate places in the TTPP proof rules, or rather, the extension of these rules studied by Owicki and Gries [20]. Although the mere insertion of the separating conjunction was straightforward, we found we could handle a number of daring, though valuable, programming idioms, and this opened up a number of unexpected (for us) possibilities.

To describe the nature of the daring programs we suppose that there is a way in the programming language to express groupings of mutual exclusion. A "mutual exclusion group" is a class of commands whose elements (or their occurrences) are required not to overlap in their executions. Notice that there is no requirement of atomicity; execution of commands from a mutual exclusion group might very well overlap with execution of a command not in that group. In monitor-based concurrency each monitor determines a mutual exclusion group, consisting of all calls to the monitor procedures. When programming with semaphores each semaphore s determines a group, the pair of the semaphore operations P(s) and V(s). In TTPP the collection of conditional critical regions with r when B do C with common resource name r forms a mutual exclusion group. With this terminology we may now state one of the crucial distinctions in the paper.

A program is *cautious* if, whenever concurrent processes access the same piece of state, they do so only within commands from the same mutual exclusion group. Otherwise, the program is *daring*.

Obviously, the nature of mutual exclusion is to guarantee that cautious programs are not *racy*, where concurrent processes attempt to access the same portion of state at the same time without explicit synchronization. The simplicity and modularity of the TTPP proof rules is achieved by syntactic restrictions which ensure caution; a main contribution of this paper is to take the method into the realm of daring programs, while maintaining its modular nature.

Daring programs are many. Examples include: double-buffered I/O, such as where one process renders an image represented in a buffer while a second process is filling a second buffer, and the two buffers are switched when an image changes; efficient message passing, where a pointer is passed from one process to another to avoid redundant copying of large pieces of data; memory managers and other resource managers such as thread and connection pools, which are used to avoid the overhead of creating and destroying threads or connections to databases. Indeed, almost all concurrent systems programs are daring, such as microkernel OS designs, programs that manage network connectivity and routing, and even many application programs such as web servers.

But to be daring is to court danger: If processes access the same portion of state outside a common mutual exclusion grouping then they just might do so at

51

the same time, and we can very well get inconsistent results. Yet it is possible to be safe, and to know it, when a program design observes a principle of resource separation.

Separation Property. At any time, the state can be partitioned into that "owned" by each process and each mutual exclusion group.

When combined with the principle that a program component only accesses state that it owns, separation implies race-freedom.

Our proof system will be designed to ensure that any program that gets past the proof rules satisfies the Separation Property. And because we use a logical connective (the separating conjunction) rather than scoping constraints to express separation, we are able to describe dynamically changing state partitions, where ownership (the right to access) transfers between program components. It is this that takes us safely into the territory of daring programs.

This paper is very much about fluency with the logic – how to reason with it – rather than its metatheory; we refer the reader to the companion paper by Stephen Brookes for a thorough theoretical analysis [4]. In addition to soundness, Brookes shows that any proven program will not have a race in an execution starting from a state satisfying its precondition.

After describing the proof rules we give two examples, one of a pointertransferring buffer and the other of a toy memory manager. These examples are then combined to illustrate the modularity aspect. The point we will attempt to demonstrate is that the specification for each program component is "local" or "self contained", in the sense that assertions make local remarks about the portions of state used by program components, instead of global remarks about the entire system state. Local specification and reasoning is essential if we are ever to have reasoning methods that scale; of course, readers will have to judge for themselves whether the specifications meet this aim.

This is a preliminary paper. In the long version we include several further examples, including two semaphore programs and a proof of parallel mergesort.

2 The Programming Language

The presentation of the programming language and the proof rules in this section and the next follows that of Owicki and Gries [20], with alterations to account for the heap. As there, we will concentrate on programs of a special form, where we have a single resource declaration, possibly prefixed by a sequence of assignments to variables, and a single parallel composition of sequential commands.

```
init;
resource r_1(variable list), ..., r_m(variable list)
C_1 \parallel \cdots \parallel C_n
```

It is possible to consider nested resource declarations and parallel compositions, but the basic case will allow us to describe variable side conditions briefly

Table	1.	Sequential	Command	ls
-------	----	------------	---------	----

 $\begin{array}{ll} C & ::= x := E \mid x := [E] \mid [E] := F \mid x := \operatorname{cons}(E_1, ..., E_n) \mid \operatorname{dispose}(E) \\ & \mid & \operatorname{skip} \mid C; C \mid \operatorname{if} B \operatorname{then} C \operatorname{else} C \mid \operatorname{while} B \operatorname{do} C \\ & \mid & \operatorname{with} r \operatorname{when} B \operatorname{do} C \end{array}$ $E, F ::= x, y, ... \mid 0 \mid 1 \mid E + F \mid E \times F \mid E - F$ $B & ::= \operatorname{false} \mid B \Rightarrow B \mid E = F \mid E < F \end{array}$

in an old-fashioned, wordy style. We restrict to this basic case mainly to get more quickly to examples and the main point of this paper, which is exploration of idioms (fluency). We refer to [4] for a more modern presentation of the programming language, which does not observe this restricted form.

A grammar for the sequential processes is included in Table 1. They include constructs for while programs as well as operators for accessing a program heap. The operations [E] := F and x := [E] are for mutating and reading heap cells, and the commands $x := \operatorname{cons}(E_1, ..., E_n)$ and dispose(E) are for allocating and deleting cells. Note that the integer expressions E are pure, in that they do not themselves contain any heap dereferencing [·]. Also, although expressions range over arbitrary integers, the heap is addressed by non-negative integers only; the negative numbers can be used to represent data apart from the addresses, such as atoms and truth values, and we will do this without comment in examples like in Section 4 where we include true, false and nil amongst the expressions E (meaning, say, -1, -2 and -3).

The command for accessing a resource is the conditional critical region:

with r when B do C.

Here, *B* ranges over (heap independent) boolean expressions and *C* over commands. Each resource name determines a mutual exclusion group: two with commands for the same resource name cannot overlap in their executions. Execution of with r when B do C can proceed if no other region for r is currently executing, and if the boolean condition B is true; otherwise, it must wait until the conditions for it to proceed are fulfilled.

It would have been possible to found our study on monitors rather than CCRs, but this would require us to include a procedure mechanism and it is theoretically simpler not to do so.

Programs are subject to variable conditions for their well-formedness (from [20]). We say that a variable *belongs to* resource r if it is in the associated variable list in a resource declaration. We require that

- 1. a variable belongs to at most one resource;
- 2. if variable x belongs to resource r, it cannot appear in a parallel process except in a critical region for r; and
- 3. if variable x is changed in one process, it cannot appear in another unless it belongs to a resource.

53

SYNTAX

$$P, Q, R ::= B \mid \text{emp} \mid E \mapsto F \mid P * Q \mid \text{false} \mid P \Rightarrow Q \mid \forall x.P \mid \cdots$$

ABBREVIATIONS

 $\neg P = P \Rightarrow \texttt{false}; \texttt{true} \stackrel{\Delta}{=} \neg(\texttt{false}); P \lor Q \stackrel{\Delta}{=} (\neg P) \Rightarrow Q; P \land Q \stackrel{\Delta}{=} \neg(\neg P \lor \neg Q); \exists x. P \stackrel{\Delta}{=} \neg \forall x. \neg P$ $E \mapsto F_0, \dots, F_n \stackrel{\Delta}{=} (E \mapsto F_0) \ast \cdots \ast (E + n \mapsto F_n)$ $E \mapsto - \stackrel{\Delta}{=} \exists y. E \mapsto y \qquad (y \notin \text{Free}(E))$

For the third condition note that a variable x is changed by an assignment command x := -, but not by [x] := E; in the latter it is a heap cell, rather than a variable, that is altered.

These conditions ensure that any variables accessed in two concurrent processes must be protected by synchronization. For example, the racy program

 $x := 3 \parallel x := x + 1$

is ruled out by the conditions. In the presence of pointers these syntactic restrictions are not enough to avoid all races. In the legal program

 $[x] := 3 \parallel [y] := 4$

if x and y denote the same integer in the starting state then they will be aliases and we will have a race, while if x and y are unequal then there will be no race.

3 Proof Rules

The proof rules below refer to assertions from separation logic; see Table 2. The assertions include the points-to relation $E \mapsto F$, the separating conjunction *, the empty-heap predicate emp, and all of classical logic. The use of \cdots in the grammar means we are being open-ended, in that we allow for the possibility of other forms such as the -* connective from BI or a predicate for describing linked lists, as in Section 5. A semantics for these assertions has been included in the appendix.

Familiarity with the basics of separation logic is assumed [22]. For now we only remind the reader of two main points. First, P * Q means that the (current, or owned) heap can be split into two components, one of which makes P true and the other of which makes Q true. Second, to reason about a dereferencing operation we must know that a cell exists in a precondition. For instance, if $\{P\}[10] := 42\{Q\}$ holds, where the statement mutates address 10, then P must imply the assertion $(10 \mapsto -) * true$ that 10 not be dangling. Thus, a precondition

confers the right to access certain cells, those that it guarantees are not dangling; this provides the connection between program logic and the intuitive notion of "ownership" discussed in the introduction.

```
To reason about a program

init;

resource r_1(variable list), ..., r_m(variable list)

C_1 \parallel \cdots \parallel C_n
```

we first specify a formula RI_{r_i} , the resource invariant, for each resource name r_i . These formulae must satisfy

- any command $x := \cdots$ changing a variable x which is free in RI_{r_i} must occur within a critical region for r_i .

Owicki and Gries used a stronger condition, requiring that each variable free in RI_{r_i} belong to resource r_i . The weaker condition is due to Brookes, and allows a resource invariant to connect the value of a protected variable with the value of an unprotected one.

Also, for soundness we need to require that each resource invariant is "precise". The definition of precision, and an example of Reynolds showing the need to restrict the resource invariants, is postponed to Section 7; for now we will just say that the invariants we use in examples will adhere to the restriction.

In a complete program the resource invariants must be *separately* established by the initialization sequence, together with an additional portion of state that is given to the parallel processes for access outside of critical regions. The resource invariants are then removed from the pieces of state accessed directly by processes. This is embodied in the

RULE FOR COMPLETE PROGRAMS
$$\frac{\{P\}init\{RI_{r_1} * \cdots * RI_{r_m} * P'\}}{\{P\}} \qquad \{P'\}C_1 \parallel \cdots \parallel C_n\{Q\}}$$

$$\frac{\{P\}}{init;}$$
resource r_1 (variable list), ..., r_m (variable list)
$$C_1 \parallel \cdots \parallel C_n$$

$$\{RI_{r_1} * \cdots * RI_{r_m} * Q\}$$

For a parallel composition we simply give each process a separate piece of state, and separately combine the postconditions for each process.

PARALLEL COMPOSITION RULE

$$\frac{\{P_1\}C_1\{Q_1\}\cdots\{P_n\}C_n\{Q_n\}}{\{P_1*\cdots*P_n\}C_1\parallel\cdots\parallel C_n\{Q_1*\cdots*Q_n\}} \text{ no variable free in } P_i \text{ or } Q_i \text{ is changed in } C_j \text{ when } j \neq i$$

Using this proof rule we can prove a program that has a potential race, as long as that race is ruled out by the precondition.

$$\frac{\{x \mapsto 3\} [x] := 4 \{x \mapsto 4\} \qquad \{y \mapsto 3\} [y] := 5 \{y \mapsto 5\}}{\{x \mapsto 3 * y \mapsto 3\} [x] := 4 \parallel [y] := 5 \{x \mapsto 4 * y \mapsto 5\}}$$

Here, the * in the precondition guarantees that x and y are not aliases.

It will be helpful to have an annotation notation for (the binary case of) the parallel composition rule. We will use an annotation form where the overall precondition and postcondition come first and last, vertically, and are broken up for the annotated constituent processes; so the just-given proof is pictured

$$\begin{array}{c} \{x \mapsto 3 * y \mapsto 3\} \\ \{x \mapsto 3\} & \{y \mapsto 3\} \\ [x] := 4 & \parallel & [y] := 5 \\ \{x \mapsto 4\} & \{y \mapsto 5\} \\ \{x \mapsto 4 * y \mapsto 5\} \end{array}$$

The reasoning that establishes the triples $\{P_j\}C_j\{Q_j\}$ for sequential processes in the parallel rule is done in the context of an assignment of invariants RI_{r_i} to resource names r_i . This contextual assumption is used in the

CRITICAL REGION RULE

$$\frac{\{(P * RI_r) \land B\} C \{Q * RI_r\}}{\{P\} \text{ with } r \text{ when } B \text{ do } C \{Q\}} \text{ variables free in } P \text{ or } Q$$

The idea of this rule is that when inside a critical region the code gets to see the state associated with the resource name as well as that local to the process it is part of, while when outside the region reasoning proceeds without knowledge of the resource's state.

The side condition "No other process..." refers to the form of a program as composed of a fixed number of processes $C_1 \parallel \cdots \parallel C_n$, where an occurrence of a with command will be in one of these processes C_j .

Besides these proof rules we allow all of sequential separation logic; see the appendix. The soundness of proof rules for sequential constructs is delicate in the presence of concurrency. For instance, we can readily derive

 $\{10 \mapsto 3\}x := [10]; x := [10]\{(10 \mapsto 3) \land x = 3\}$

in separation logic, but if there was interference from another process, say altering the contents of 10 between the first and second statements, then the triple would not be true.

The essential point is that proofs in our system build in the assumption that there is "no interference from the outside", in that processes only affect one another at explicit synchronization points. This mirrors a classic program design principle of Dijkstra, that "apart from the (rare) moments of explicit intercommunication, the individual processes are to be regarded as completely independent of each other" [8]. It allows us to ignore the minute details of potential interleavings of sequential programming constructs, thus greatly reducing the number of process interactions that must be accounted for in a verification.

In sloganeering terms we might say that *well specified processes mind their own business:* proven processes only dereference those cells that they own, those known to exist in a precondition for a program point. This, combined with the use of * to partition program states, implements Dijkstra's principle.

These intuitive statements about interference and ownership receive formal underpinning in Brookes's semantic model [4]. The most remarkable part of his analysis is an interplay between an interleaving semantics based on traces of actions and a "local enabling" relation that "executes" a trace in a portion of state owned by a process. The enabling relation skips over intermediate states and explains the "no interference from the outside" idea.

4 Example: Pointer-Transferring Buffer

For efficient message passing it is often better to pass a pointer to a value from one process to another, rather than passing the value itself; this avoids unneeded copying of data. For example, in packet-processing systems a packet is written to storage by one process, which then inserts a pointer to the packet into a message queue. The receiving process, after finishing with the packet, returns the pointer to a pool for subsequent reuse. Similarly, if a large file is to be transmitted from one process to another it can be better to pass a pointer than to copy its contents. This section considers a pared-down version of this scenario, using a one-place buffer.

In this section we use operations cons and dispose for allocating and deleting *binary* cons cells. (To be more literal, dispose(E) in this section would be expanded into dispose(E); dispose(E + 1) in the syntax of Section 2.)

The initialization and resource declaration are

```
full := false;
resource buf(c, full)
```

and we have code for putting a value into the buffer and for reading it out.

```
put(x) \stackrel{\Delta}{=} with buf when \neg full do
c := x; full := true;
```

```
get(y) \stackrel{\Delta}{=} with buf when full do
 <math>y := c; full := false;
```

For presentational convenience we are using definitions of the form

```
name(x) \stackrel{\Delta}{=} with r when B do C
```

to encapsulate operations on a resource. In this we are not introducing a procedure mechanism, but are merely using name(x) as an abbreviation.

We focus on the following code.

$x := \operatorname{cons}(a, b);$	$\mathtt{get}(y);$
$\mathtt{put}(x);$	use(y);
	dispose(y):

This creates a new pointer in one process, which points to a binary cons cell containing values a and b. To transmit these values to the other process, instead

57

of copying both a and b the pointer itself is placed in the buffer. The second process reads the pointer out, uses it in some way, and finally disposes it. To reason about the dispose operation in the second process, we must ensure that $y \mapsto -, -$ holds beforehand. At the end of the section we will place these code snippets into loops, as part of a producer/consumer iidiom, but for now will concentrate on the snippets themselves.

The resource invariant for the buffer is

$$RI_{buf}$$
: $(full \land c \mapsto -, -) \lor (\neg full \land emp).$

To understand this invariant it helps to use the "ownership" or "permission" reading of separation logic, where an assertion P at a program point implies that "I have the right to dereference the cells in P here", or more briefly, "I own P" [18]. According to this reading the assertion $c \mapsto -, -$ says "I own binary cons cell c" (and I don't own anything else). The assertion emp does not say that the global state is empty, but rather that "I don't own any heap cells, here". Given this reading the resource invariant says that the buffer owns the binary cons cell associated with c when *full* is true, and otherwise it owns no heap cells.

Here is a proof for the body of the with command in put(x).

$$\begin{array}{l} \{(RI_{buf} * x \mapsto -, -) \land \neg full\} \\ \{(\neg full \land emp) * x \mapsto -, -\} \\ \{x \mapsto -, -\} \\ c := x; full := true \\ \{full \land c \mapsto -, -\} \\ \{RI_{buf}\} \\ \{RI_{buf} * emp\} \end{array}$$

The rule for with commands then gives us

 ${x \mapsto -, -}$ put(x) {emp}.

The postcondition indicates that the sending process gives up ownership of pointer x when it is placed into the buffer, even though the value of x is still held by the sender.

A crucial point in the proof of the body is the implication

 $full \land c \mapsto -, - \Rightarrow RI_{buf}$

which is applied in the penultimate step. This step reflects the idea that the knowledge "x points to something" flows out of the user program and into the buffer resource. On exit from the critical region x does indeed point to something in the global state, but this information cannot be recorded in the postcondition of put. The reason is that we used $c \mapsto -, -$ to re-establish the resource invariant; having $x \mapsto -, -$ as the postcondition would be tantamount to asserting $(x \mapsto -, -) * (c \mapsto -, -)$ at the end of the body of the with command, and this assertion is necessarily false when c and x are equal, as they are at that point.

The flipside of the first process giving up ownership is the second's assumption of it:
$$\begin{array}{l} \{(RI_{buf}*\texttt{emp}) \land full\} \\ \{full \land c \mapsto -, -\} \\ y := c; \ full := \texttt{false} \\ \{y \mapsto -, - \land \neg full\} \\ \{(\neg full \land \texttt{emp}) * y \mapsto -, -\} \\ \{RI_{buf}* y \mapsto -, -\}, \end{array}$$

which gives us

```
\{\texttt{emp}\}\texttt{get}(y)\{y\mapsto -,-\}.
```

We can then prove the parallel processes as follows, assuming that use(y) satisfies the indicated triple.

$$\begin{array}{ll} \{ \texttt{emp} \ast \texttt{emp} \} & \{ \texttt{emp} \} & \\ \{ \texttt{emp} \} & & \{ \texttt{emp} \} \\ x := \texttt{cons}(a,b); & \parallel & \texttt{get}(y); \\ \{ x \mapsto -, - \} & & \{ y \mapsto -, - \} \\ \texttt{put}(x); & & use(y); \\ \{ \texttt{emp} \} & & \{ y \mapsto -, - \} \\ & & \texttt{dispose}(y); \\ \{ \texttt{emp} \} & \\ \{ \texttt{emp} \} & \\ \{ \texttt{emp} \} \end{array}$$

Then using the fact that the initialization establishes the resource invariant in a way that gets us ready for the parallel rule

$$\{ emp \}$$

full := false
 $\{ \neg full \land emp \}$
 $\{ RI_{buf} * emp * emp \}$

we obtain the triple $\{emp\}prog\{RI_{buf}\}$ for the complete program prog.

In writing annotated programs we generally include assertions at program points to show the important properties that hold; to formally connect to the proof theory we would sometimes have to apply an axiom followed by the Hoare rule of consequence or other structural rules. For instance, in the left process above we used $x \mapsto -, -$ as the postcondition of $x := \operatorname{cons}(a, b)$; to get there from the "official" postcondition $x \mapsto a, b$ we just observe that it implies $x \mapsto -, -$. We will often omit mention of little implications such as this one.

The verification just given also shows that if we were to add a command, say x.1 := 3, that dereferences x after the put command in the left process then we would not be able to prove the resulting program. The reason is that emp is the postcondition of put(x), while separation logic requires that x point to something (be owned) in the precondition of any operation that dereferences x.

In this verification we have concentrated on tracking ownership, using assertions that are type-like in nature: they say what kind of data exists at various

Table 3. Pointer-passing Producer/Consumer Program

```
{emp}
                full := false;
                \{ emp \land \neg full \}
                \{RI_{buf} * emp * emp\}
               resource buf(c, full)
                    {emp * emp}
{emp}
                                         {emp}
while true do
                                         while true do
       {emp}
                                                {emp}
       produce(a, b);
                                                get(y);
      x := \operatorname{cons}(a, b);
                               1
                                                use(y);
      put(x);
                                                dispose(y);
       {emp}
                                                {emp}
{false}
                                         {false}
                   {false * false}
               \{RI_{huf} * \texttt{false}\}
               {false}
```

program points, but do not speak of the identities of the data. For instance, because the assertions use -, - they do not track the flow of the values a and b from the left to the right process. To show stronger correctness properties, which track buffer contents, we would generally need to use auxiliary variables [20].

As it stands the code we have proven is completely sequential: the left process must go first. Using the properties we have shown it is straightforward to prove a producer/consumer program, where these code snippets are parts of loops, as in Table 3. In the code there emp is the invariant for each loop, and the overall property proven ensures that there is no race condition.

5 Example: Memory Manager

A resource manager keeps track of a pool of resources, which are given to requesting processes, and received back for reallocation. As an example of this we consider a toy manager, where the resources are memory chunks of size two. The manager maintains a free list, which is a singly-linked list of binary cons cells. The free list is pointed to by f, which is part of the declaration

```
resource mm(f).
```

The invariant for mm is just that f points to a singly-linked list without any dangling pointers in the link fields:

 RI_{mm} : list f.

The *list* predicate is the least satisfying the following recursive specification.

$$listx \iff (x = nil \land emp) \lor (\exists y. x \mapsto -, y * listy)$$

When a user program asks for a new cell, *mm* gives it a pointer to the first element of the free list, if the list is nonempty. In case the list is empty the *mm* calls cons to get an extra element.

$$alloc(x, a, b) \stackrel{\Delta}{=} with mm$$
 when true do
if $f = nil$ then $x := cons(a, b)$
 $else \ x := f; \ f := x.2; \ x.1 := a; \ x.2 := b$
 $dealloc(y) \stackrel{\Delta}{=} with mm$ when true do

dealloc(y) = with mm when true do y.2 := f;f := y;

The command f := x.2 reads the cdr of binary cons cell x and places it into f. We can desugar x.2 as [x + 1] in the RAM model of separation logic, and similarly we will use x.1 for [x] to access the car of a cons cell.

Using the rule for with commands we obtain the following "interface specifications":

 $\{ emp \} alloc(x, a, b) \{ x \mapsto a, b \}$ $\{ y \mapsto -, - \} dealloc(y) \{ emp \}.$

The specification of alloc(x, a, b) illustrates how ownership of a pointer materializes in the user code, for subsequent use. Conversely, the specification of dealloc requires ownership to be given up. The proofs of the bodies of these operations using the with rule describe ownership transfer in much the same way as in the previous section, and are omitted.

Since we have used a critical region to protect the free list from corruption, it should be possible to have parallel processes that interact with *mm*. A tiny example of this is just two processes, each of which allocates, mutates, then deallocates.

$$\{ emp * emp \}$$

$$\{ emp \}$$

$$alloc(x, a, b);$$

$$\{ x \mapsto a, b \}$$

$$x.1 := 4$$

$$y.1 := 7$$

$$\{ x \mapsto 4, b \}$$

$$\{ y \mapsto 7, b' \}$$

$$dealloc(x);$$

$$\{ emp \}$$

$$\{ emp \}$$

$$\{ emp \}$$

This little program is an example of one that is daring but still safe. To see the daring aspect, consider an execution where the left process goes first, right up to completion, before the right one begins. Then the statements mutating x.1 and y.1 will in fact alter the same cell, and these statements are not within

critical regions. However, although there is potential aliasing between x and y, the program proof tells us that there is no possibility of racing in *any* execution.

On the other hand, if we were to insert a command x.1 := 8 immediately following dealloc(x) in the leftmost process then we would indeed have a race. However, the resulting program would not get past the proof rules, because the postcondition of dealloc(x) is emp.

The issue here is not exclusive to memory managers. When using a connection pool or a thread pool in a web server, for example, once a handle is returned to the pool the returning process must make sure not to use it again, or inconsistent results may ensue.

6 Combining the Buffer and Memory Manager

We now show how to put the treatment of the buffer together with the homegrown memory manager *mm*, using alloc and dealloc instead of cons and dispose. The aim is to show different resources interacting in a modular way.

We presume now that we have the resource declarations for both *mm* and *buf*, and their associated resource invariants. Here is the proof for the parallel processes in Section 4 done again, this time using mm.

$$\{ emp * emp \}$$

$$\{ emp \}$$

$$\{ emp \}$$

$$alloc(x, a, b); \parallel$$

$$\{ x \mapsto -, - \}$$

$$y \mapsto -, - \}$$

$$put(x);$$

$$use(y);$$

$$\{ emp \}$$

$$\{ emp * emp \}$$

$$\{ emp \}$$

In this code, a pointer's ownership is first transferred out of the *mm* resource into the lefthand user process. It then gets sent into the *buf* resource, from where it taken out by the righthand process and promptly returned to *mm*.

The initialization sequence and resource declaration now have the form

```
full := false;
resource buf(c, full), mm(f)
```

and we have the triple

```
\{\texttt{list}(f)\} full := false \{RI_{buf} * RI_{mm} * \texttt{emp} * \texttt{emp}\}
```

which sets us up for reasoning about the parallel composition. We can use the rule for complete programs to obtain a property of the complete program.

The point is that we did not have to change any of the code or verifications done with *mm* or with *buf* inside the parallel processes; we just used the same preconditions and postconditions for get, put, alloc and dealloc, as given to

us by the proof rule for CCRs. The crucial point is that the rule for CCRs does not include the resource invariant in the "interface specification" described by the conclusion of the rule. As a result, a proof using these specifications does not need to be repeated, even if we change the implementation and internal resource invariant of a module. Effective resource separation allows us to present a localized view, where the state of a resource is hidden from user programs (when outside critical regions).

7 The Reynolds Counterexample

The following counterexample, due to John Reynolds, shows that the concurrency proof rules are incompatible with the usual Hoare logic rule of conjunction

$$\frac{\{P\}C\{Q\} \quad \{P'\}C\{Q'\}}{\{P \land P'\}C\{Q \land Q'\}}$$

The example uses a resource declaration

resource r()

with invariant

 $RI_r = true.$

Let one stand for the assertion $10 \mapsto -$. First, we have the following derivation using the axiom for skip, the rule of consequence, and the rule for critical regions.

 $\frac{\{\texttt{true}\}\texttt{skip}\{\texttt{true}\}}{\{(\texttt{emp} \lor \texttt{one}) * \texttt{true}\}\texttt{skip}\{\texttt{emp} * \texttt{true}\}}}{\{\texttt{emp} \lor \texttt{one}\}\texttt{with } r \texttt{ when true do skip} \{\texttt{emp}\}}$

Then, from the conclusion of this proof, we can construct two derivations:

 $\frac{\{\operatorname{emp} \lor \operatorname{one}\} \operatorname{with} r \text{ when true do skip } \{\operatorname{emp}\}}{\{\operatorname{emp}\} \operatorname{with} r \operatorname{when true do skip } \{\operatorname{emp}\}}$ $\frac{\{\operatorname{emp} \ast \operatorname{one}\} \operatorname{with} r \operatorname{when true do skip } \{\operatorname{emp} \ast \operatorname{one}\}}{\{\operatorname{one}\} \operatorname{with} r \operatorname{when true do skip } \{\operatorname{one}\}}$

and

```
\frac{\{\texttt{emp} \lor \texttt{one}\} \texttt{with } r \texttt{ when true do skip } \{\texttt{emp}\}}{\{\texttt{one}\} \texttt{with } r \texttt{ when true do skip } \{\texttt{emp}\}}
```

Both derivations begin with the rule of consequence, using the implications $emp \Rightarrow emp \lor one$ and $one \Rightarrow emp \lor one$. The first derivation continues with an application of the ordinary frame rule, with invariant one, and one further use of consequence.

The conclusions of these two derivations are incompatible with one another. The first says that ownership of the single cell is kept by the user code, while

the second says that it is swallowed up by the resource. An application of the conjunction rule with these two conclusions gives us the premise of the following which, using the rule of consequence, leads to an inconsistency.

```
\frac{\{\texttt{one} \land \texttt{one}\} \texttt{with } r \texttt{ when true do skip } \{\texttt{emp} \land \texttt{one}\}}{\{\texttt{one}\} \texttt{with } r \texttt{ when true do skip } \{\texttt{false}\}}
```

The last triple would indicate that the program diverges, where it clearly does not.

The fact that the resource invariant true does not precisely say what storage is owned conspires together with the nondeterministic nature of * to fool the proof rules. A way out of this problem is to insist that resource invariants precisely nail down a definite area of storage [18]. In the semantic notation of the appendix,

an assertion *P* is *precise* if for all states (s, h) there is at most one subheap $h' \subseteq h$ where $s, h' \models P$.

The subheap h' here is the area of storage that a precise predicate identifies.

The Reynolds counterexample was discovered in August of 2002, a year after the author had described the proof rules and given the pointer-transferring buffer example in an unpublished note. Realizing that the difficulty in the example had as much to do with information hiding as concurrency, the author, Yang and Reynolds studied a version of the problem in a sequential setting, where precise resource invariants were used to describe the internal state of a module [18]. The more difficult concurrent case was then settled by Brookes [4]; his main result is

Theorem (Brookes): the proof rules are sound if all resource invariants are precise predicates.

This rules out Reynolds's counterexample because true is not a precise predicate. And the resource invariants in the one-place buffer and the toy memory manager are both precise.

8 Conclusion

It may seem as if the intuitive points about separation made in this paper should apply more generally than to shared-variable concurrency; in particular, it would be interesting to attempt to provide modular methods for reasoning about process calculi using resource-oriented logics. In CSP the concepts of resource separation and sharing have been modelled in a much more abstract way than in this paper [13]. And the π -calculus is based on very powerful primitives for name manipulation [15], which are certainly reminiscent of pointers in imperative programs. In both cases it is natural to wonder whether one could have a logic which allows names to be successively owned by different program components, while maintaining the resource separation that is often the basis of system designs. However, the right way of extending the ideas here to process calculi is not obvious.

A line of work that bears a formal similarity to ours is that of Caires, Cardelli and Gordon on logics for process calculi [6,5]. Like here, they use a mixture of substructural logic and ordinary classical logic and, like here, they consider concurrency. But independence between processes has not been emphasized in their work – there is no analogue of what we called the Separation Property – and neither have they considered the impact of race conditions. Their focus is instead on the expression of what they call "intensional" properties, such as the number of connections between two processes. So, although similar in underlying logical technology, their approach uses this technology in a very different way.

The idea of ownership is, as one might expect, central in work on Ownership Types [7]. It would be interesting to attempt to describe a formal connection.

Stepping back in time, one of the important early works on reasoning about imperative concurrent programs was that of Owicki and Gries [19]. A difference with the work here is that our system rules out racy programs, while theirs does not. However, they handle racy programs by assuming a fixed level of granularity, where if we were to make such an assumption explicit (using a critical region) such programs would not be, in principle, out of reach of our methods. More importantly, the Owicki-Gries method involves explicit checking of non-interference between program components, while our system rules out interference in an implicit way, by the nature of the way that proofs are constructed. The result is that the method here is more modular.

This last claim is not controversial; it just echoes a statement of Owicki and Gries. There are in fact two classic Owicki-Gries works, one [20] which extends the approach of Hoare in TTPP, and another [19] which is more powerful but which involves explicit non-interference checking. They candidly acknowledge that "the proof process becomes much longer" in their more powerful method; one way to view this work is as an attempt to extend the more modular of the two approaches, where the proof process is shorter, to a wider variety of programs.

There are a number of immediate directions for future work. One is the incorporation of passivity, which would allow read-only sharing of heap cells between processes. Another is proof methods that do not require complete resource separation, such as the rely-guarantee method [14,23], where the aim would be to use separation logic's local nature to cut down the sizes of rely and guarantee conditions. A third is the incorporation of temporal features. Generally, however, we believe that the direction of resource-oriented logics offers promise for reasoning about concurrent systems, as we hope to have demonstrated in the form of proofs and specifications given in this paper.

Acknowledgements. I am grateful to Per Brinch Hansen, David Pym and John Reynolds for discussions on resource and concurrency that impacted the form of this work, to Steve Brookes for the theoretical analysis without which it could not be presented, and to Josh Berdine, Richard Bornat, Cristiano Calcagno and Hongseok Yang for daring me to embrace the daring programming idioms. This research was supported by the EPSRC.

References

- 1. P. Brinch Hansen. The nucleus of a multiprogramming system. *Comm. ACM*, 13(4):238–250, 1970.
- 2. P. Brinch Hansen. Structured multiprogramming. *Comm. ACM*, 15(7):574–578, 1972. Reprinted in [3].
- 3. P. Brinch Hansen, editor. *The Origin of Concurrent Programming*. Springer-Verlag, 2002.
- 4. S. D. Brookes. A semantics for concurrent separation logic. This Volume, Springer LNCS, *Proceedings of the 15th CONCUR*, London. August, 2004.
- L. Cardelli and L Caires. A spatial logic for concurrency. In 4th International Symposium on Theoretical Aspects of Computer Science, LNCS 2255:1–37, Springer, 2001.
- 6. L. Cardelli and A. D. Gordon. Anytime, anywhere. modal logics for mobile ambients. In 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 365–377, 2000.
- D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 53-76, Springer LNCS 2072, 2001.
- E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968. Reprinted in [3].
- 9. E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1 2:115–138, October 1971. Reprinted in [3].
- 10. J.-Y. Girard. Linear logic. Theoretical Computer Science, 50:1-102, 1987.
- 11. C. A. R. Hoare. Towards a theory of parallel programming. In Hoare and Perrot, editors, *Operating Systems Techniques*. Academic Press, 1972. Reprinted in [3].
- 12. C. A. R. Hoare. Monitors: An operating system structuring concept. *Comm. ACM*, 17(10):549–557, 1974. Reprinted in [3].
- 13. C. A. R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985.
- 14. C. B. Jones. Specification and design of (parallel) programs. *IFIP Conference*, 1983.
- R. Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.
- P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of 15th Annual Conference of the European Association for Computer Science Logic*, LNCS, pages 1–19. Springer-Verlag, 2001.
- P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 99.
- P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 268–280, Venice, January 2004.
- 19. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, (19):319–340, 1976.
- S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. Comm. ACM, 19(5):279–285, 1976.
- 21. A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1), 45–60, 1981.

- J. C. Reynolds. Separation logic: a logic for shared mutable data structures. Invited Paper, *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 55-74, 2002.
- C. Stirling. A generalization of the Owicki-Gries Hoare logic for a concurrent while language. *Theoretical Computer Science*, 58:347–359, 1988.

Appendix: Sequential Separation Logic

Reasoning about atomic commands is based on the "small axioms" where x, m, n are assumed to be distinct variables.

$$\begin{array}{l} \{E \mapsto -\} [E] := F \left\{ E \mapsto F \right\} \\ \{E \mapsto -\} \operatorname{dispose}(E) \left\{ \operatorname{emp} \right\} \\ \{x = m \wedge \operatorname{emp} \} x := \operatorname{cons}(E_1, ..., E_k) \{x \mapsto E_1[m/x], ..., E_k[m/x]\} \\ \{x = n \wedge \operatorname{emp} \} x := E \left\{ x = (E[n/x]) \wedge \operatorname{emp} \right\} \\ \{E \mapsto n \wedge x = m\} x := [E] \left\{ x = n \wedge E[m/x] \mapsto n \right\} \end{array}$$

Typically, the effects of these "small" axioms can be extended using the frame rule:

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}} C \text{ doesn't change}$$
variables free in R

In addition to the above we have the usual proof rules of standard Hoare logic.

$$\frac{\{P \land B\}C\{P\}}{\{P\}\text{while } B \text{ do } C\{P \land \neg B\}} \qquad \frac{P \Rightarrow P' \quad \{P'\}C\{Q'\} \quad Q' \Rightarrow Q}{\{P\}C\{Q\}}$$

$$\frac{\{P\}\text{cl}\{Q\} \quad \{Q\}C_2\{R\}}{\{P\}\text{cl}\{Q\}}$$

$$\frac{\{P \land B\}C\{Q\} \quad \{P \land \neg B\}C'\{Q\}}{\{P\}\text{ if } B \text{ then } C \text{ else } C'\{Q\}}$$

Also, although we have not stated them, there is a substitution rule and a rule for introducing existential quantifiers, as in [16].

We can use $P \Rightarrow Q$ in the consequence rule when $s, h \models P \Rightarrow Q$ holds for all s and h in the semantics below (when the domain of s contains the free variables of P and Q.) Thus, the semantics is, in this paper, used as an oracle by the proof system.

A state consists of two components, the stack $s \in S$ and the heap $h \in H$, both of which are finite partial functions as indicated in the following domains.

Variables
$$\stackrel{\Delta}{=} \{x, y, ...\}$$
Nats $\stackrel{\Delta}{=} \{0, 1, 2...\}$ Ints $\stackrel{\Delta}{=} \{..., -1, 0, 1, ...\}$ $H \stackrel{\Delta}{=} Nats \rightarrow_{fin} Ints$ $S \stackrel{\Delta}{=} Variables \rightarrow_{fin} Ints$ States $\stackrel{\Delta}{=} S \times H$

Integer and boolean expressions are determined by valuations

$$\llbracket E \rrbracket s \in \texttt{Ints}$$
 $\llbracket B \rrbracket s \in \{true, false\}$

where the domain of $s \in S$ includes the free variables of *E* or *B*. We use the following notations in the semantics of assertions.

- dom(h) denotes the domain of definition of a heap h ∈ H, and dom(s) is the domain of s ∈ S;
- 2. h#h' indicates that the domains of h and h' are disjoint;
- 3. $h \cdot h'$ denotes the union of disjoint heaps (i.e., the union of functions with disjoint domains);
- 4. $(f \mid i \mapsto j)$ is the partial function like f except that i goes to j.

The satisfaction judgement $s, h \models P$ which says that an assertion holds for a given stack and heap. (This assumes that $Free(P) \subseteq dom(s)$, where Free(P) is the set of variables occurring freely in P.)

$$\begin{array}{ll} s,h \models B & \text{iff } \llbracket B \rrbracket s = true \\ s,h \models P \Rightarrow Q & \text{iff if } s,h \models P & \text{then } s,h \models Q \\ s,h \models \forall x.P & \text{iff } \forall v \in \text{Ints. } [s \mid x \mapsto v], h \models P \\ s,h \models \text{emp} & \text{iff } h = [] & \text{is the empty heap} \\ s,h \models E \mapsto F & \text{iff } \{\llbracket E \rrbracket s\} = dom(h) & \text{and } h(\llbracket E \rrbracket s) = \llbracket F \rrbracket s \\ s,h \models P * Q & \text{iff } \exists h_0, h_1. \ h_0 \# h_1, \ h_0 \cdot h_1 = h, \ s, h_0 \models P & \text{and } s, h_1 \models Q \end{array}$$

Notice that the semantics of $E \mapsto F$ is "exact", where it is required that E is the only active address in the current heap. Using * we can build up descriptions of larger heaps. For example, $(10 \mapsto 3) * (11 \mapsto 10)$ describes two adjacent cells whose contents are 3 and 10.

The "permissions" reading of assertions is intimately related to the way the semantics above works with "portions" of the heap. Consider, for example, a formula

$$\texttt{list}(f) * x \mapsto -, -$$

as was used in the memory manager example. A heap h satisfying this formula must have a partition $h = h_0 * h_1$ where h_0 contains the free list (and nothing else) and h_1 contains the binary cell pointed to by x. It is evident from this that we cannot regard an assertion P on its own as describing the entire state, because it might be used within another assertion, as part of a * conjunct.

Resource Control for Synchronous Cooperative Threads*

Roberto M. Amadio and Silvano Dal Zilio

Laboratoire d'Informatique Fondamentale de Marseille (LIF), CNRS and Université de Provence, France

Abstract. We develop new methods to statically bound the resources needed for the execution of systems of concurrent, interactive threads. Our study is concerned with a *synchronous* model of interaction based on cooperative threads whose execution proceeds in synchronous rounds called instants. Our contribution is a system of compositional static analyses to guarantee that each instant terminates and to bound the size of the values computed by the system as a function of the size of its parameters at the beginning of the instant.

Our method generalises an approach designed for first-order functional languages that relies on a combination of standard termination techniques for term rewriting systems and an analysis of the size of the computed values based on the notion of quasi-interpretation. These two methods can be combined to obtain an explicit polynomial bound on the resources needed for the execution of the system during an instant.

1 Introduction

The problem of bounding the usage made by programs of their resources has already attracted considerable attention. Automatic extraction of resource bounds has mainly focused on (first-order) functional languages starting from Cobham's characterisation [13] of polynomial time functions by bounded recursion on notation. Following work, see, *e.g.*, [6,14,15,16], has developed various inference techniques that allow for efficient analyses while capturing a sufficiently large range of practical algorithms.

Previous work [9,17] has shown that polynomial time or space bounds can be obtained by combining traditional termination techniques for term rewriting systems with an analysis of the size of computed values based on the notion of quasi-interpretation. Thus, in a nutshell, resource control relies on termination and bounds on data size. In [3], we have considered the problem of automatically inferring quasi-interpretations in the space of multi-variate max-plus polynomials. In [2], we have presented a virtual machine and a corresponding bytecode for a first-order functional language and shown how size and termination annotations can be formulated and verified at the level of the bytecode. In particular,

^{*} This work was partly supported by ACI Sécurité Informatique, project CRISS.

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 68-82, 2004.

[©] Springer-Verlag Berlin Heidelberg 2004

we can derive from the verification an explicit polynomial bound on the space required to execute a given bytecode.

Our approach to resource bound certification follows distinctive design decisions. First, we allow the space needed for the execution of a program to vary depending on the size of its arguments. This is in contrast to most approaches that try to enforce a constant space bound. While this latter goal is reasonable for applications targeting embedded devices, it is not always relevant in the context of mobile code. Second, our method is applicable to a large class of algorithms and does not impose specific syntactical restrictions on programs. For example, we depart from works based on a linear usage of variables [14].

Our approach to resource control should be contrasted with traditional *worst* case execution time technology (see, *e.g.*, [20]): our bounds are less precise but they apply to a larger class of algorithms and are functional in the size of the input, which seems more appropriate in the context of mobile code. In another direction, one may compare our approach with the one based on linear logic (see, *e.g.*, [11]). While in principle the linear logic approach supports higher-order functions, the approach does not offer yet a user-friendly programming language.

In this work, we aim at extending and adapting these results to a concurrent framework. Our starting point, is a quite basic and popular model of parallel threads interacting on shared variables. The kind of concurrency we consider is a *cooperative* one. This means that by default a running thread cannot be preempted unless it explicitly decides to return the control to the scheduler. In *preemptive* threads, the opposite hypothesis is made: by default a running thread can be preempted at any point unless it explicitly requires that a series of actions is atomic. We refer to, *e.g.*, [19] for an extended comparison of the cooperative model.

The second major design choice is to assume that the computation is regulated by a notion of *instant*. An instant lasts as long as a thread can make some progress in the current instant. In other terms, an instant ends when the scheduler realizes that all threads are either stopped, or waiting for the next instant, or waiting for a value that no thread can produce in the current instant. Because of this notion of instant, we regard our model as *synchronous*. Because the model includes a logical notion of time, it is possible for a thread to react to the absence of an event.

The reaction to the absence of an event, is typical of synchronous languages such as ESTEREL [8]. Boussinot *et al.* have proposed a weaker version of this feature where the reaction to the absence happens in the following instant [7] and they have implemented it in various programming environments based on C, JAVA, and SCHEME. They have also advocated the relevance of this concept for the programming of mobile code and demonstrated that the possibility for a 'synchronous' mobile agent to react to the absence of an event is an added

factor of flexibility for programs designed for open distributed systems, whose behaviours are inherently difficult to predict.

Recently, Boudol [5] has proposed a formalisation of this programming model. Our analysis will essentially focus on a small fragment of this model where higherorder functions are ruled out and dynamic thread creation, and dynamic memory allocation are only allowed at the very beginning of an instant. We believe that what is left is still expressive and challenging enough as far as resource control is concerned. Our analysis goes in three main steps. A first step is to guarantee that each instant terminates (Section 4). A second step, is to bound the size of the computed values as a function of the size of the parameters at the beginning of the instant (Section 5). A third step, is to combine the termination and size analyses. Here we show how to obtain polynomial bounds on the *space* needed for the execution of the system during an instant as a function of the size of the parameters at the beginning of the instant (Section 6). We expect that one could derive polynomial bounds on *time* as well, by adapting the work in [17].

A characteristic of our static analyses is that to a great extent they make abstraction of the memory and the scheduler. This means that each thread can be analysed separately, that the complexity of the analyses grows linearly in the number of threads, and that an incremental analysis of a dynamically changing system of threads is possible. Preliminary to these analyses, is a control flow analysis (Section 3) that guarantees that each thread reads each register at most once in an instant. We will see that without this condition, it is very easy to achieve an exponential growth of the space needed for the execution. From a technical point of view, the benefit of this *read once* condition is that it allows to regard behaviours as *functions* of their initial parameters and the registers they may read in the instant. Taking this functional viewpoint, we are able to adapt the main techniques developed for proving termination and size bounds in the first-order functional setting.

We point out that our static size analyses are not intended to predict the size of the system after arbitrary many instants. This is a harder problem which in general seems to require an understanding of the *global* behaviour of the system: typically one has to find an invariant that shows that the parameters of the system stay within certain bounds. For this reason, we believe that in practice our static analyses should be combined with a dynamic controller that at the end of each instant checks the size of the parameters of the system.

Omitted proofs may be found in a long version of this paper [1] in which we describe our programming model up to the point where a bytecode for a simple virtual machine implementing our synchronous language is defined. The long version also provides a number of programming examples illustrating how some synchronous and/or concurrent programming paradigms can be represented in our model (some simple examples are given at the end of Section 2). These examples suggest that the constraints imposed by the static analyses are not too severe and that their verification can be automated.

2 A Model of Synchronous Cooperative Threads

A *system* of synchronous cooperative threads is described by: (1) a list of mutually recursive type definitions, (2) a list of shared registers (or global variables) with a type and a default value, and (3) a list of mutually recursive functions and behaviours definitions relying on pattern matching. In this respect, the resulting programming language is reminiscent of ERLANG [4], which is a practical language to develop concurrent applications.

The set of instructions a behaviour can execute is rather minimal. Indeed, our language is already in a *pre-compiled* form where registers are assigned constant values and behaviours definitions are tail recursive. However, it is quite possible to extend the language and our analyses to have registers' names as first-class values and general recursive behaviours.

Expressions. We rely on standard notation. If α, β are formal terms then $Var(\alpha)$ is the set of free variables in α (variables in patterns are not free) and $[\alpha/x]\beta$ denotes the substitution of α for x in β . If h is a function, h[u/i] denotes a function update.

Expressions and values are built from a finite number of constructors, ranged over by c, c', \ldots We use f, f', \ldots to range over function identifiers and x, x', \ldots for variables, and distinguish the following three syntactic categories:

$v::=c(v,\ldots,v)$	(values)
$p ::= x \mid c(p, \dots, p)$	(patterns)
$e ::= x \mid c(e,\ldots,e) \mid f(e,\ldots,e)$	(expressions)

The size of an expression |e| is defined as 0 if e is a constant or a variable and $1 + \sum_{i \in 1..n} |e_i|$ if e is of the form $c(e_1, \ldots, e_n)$ or $f(e_1, \ldots, e_n)$.

A function of arity n is defined by a sequence of pattern-matching *rules* of the form $f(p_1) = be_1, \ldots, f(p_k) = be_k$, where be_i is either an expression or a thread behaviour (see below), and p_1, \ldots, p_k are sequences of length n of patterns. We follow the usual hypothesis that the patterns in p_1, \ldots, p_k are linear (a variable appears at most once). For the sake of simplicity, we will also assume that in a function definition a sequence of values v matches exactly a sequence of patterns p_i in a function definition. This hypothesis can be relaxed.

Inductive types are defined by equations of the shape $t = \cdots | \mathbf{c} of (t_1 * \cdots * t_n) | \cdots$. For instance, the type of natural numbers in unary format can be defined as follows: $nat = \mathbf{z} | \mathbf{s} of nat$. Functions, values, and expressions are assigned first order types of the shape $(t_1 * \cdots * t_n) \rightarrow t$ where t, t_1, \ldots, t_n are inductive types.

Behaviours. Some function symbols may return a thread behaviour b, b', ... rather than a value. In contrast to 'pure' expressions, a behaviour does not return a result but produces *side-effects* by reading and writing a set of global registers, ranged over by r, r', ... A behaviour may also affect the scheduling status of the thread executing it (see below).

71

```
\begin{array}{rcl} be, \dots & ::= & e \mid b \\ b, b', \dots & ::= & \operatorname{stop} \mid \operatorname{yield} b \mid f(e) \mid \operatorname{next} f(e) \mid r := e.b \mid \\ & & \operatorname{match} r \operatorname{with} p_1 \Rightarrow b_1 \mid \dots \mid p_k \Rightarrow b_k \mid [x] \Rightarrow f(e) \end{array}
```

The *effect* of the various instructions is informally described as follows: stop, terminates the executing thread for ever; yield.b, halts the execution and hands over the control to the scheduler — the control should return to the thread later in the same instant and execution resumes with b; f(e) and next.f(e) switch to another behaviour immediately or at the beginning of the following instant; r := e.b, evaluates the expression e, assigns its value to r and proceeds with the evaluation of b; match r with $p_1 \Rightarrow b_1 | \cdots | p_k \Rightarrow b_k | [x] \Rightarrow f(e)$, waits until the value of r matches one of the patterns p_1, \ldots, p_k (there could be no delay) and yields the control otherwise. At the end of the instant, if the value of r is v and no rule filters v then start the next instant with the behaviour [v/x]f(e). By convention, when the $[x] \Rightarrow \ldots$ branch is omitted, it is intended that if the match conditions are not satisfied in the current instant, then they are checked again in the following one.

Systems. Every thread has a *status*, ranged over by X, X', \ldots , that is a value in $\{N, R, S, W\}$ — where N stands for next, R for run, S for stop, and W for wait. A *system* of synchronous threads B, B', \ldots is a finite mapping from thread indexes to pairs (behaviour, status). Each register has a type and a default value — its value at the beginning of an instant — and we use s, s', \ldots to denote a *store*, an association between registers and their values. We suppose the thread indexes i, k, \ldots range over $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ and that at the beginning of each instant the store is s_o , such that each registers is assigned its default value. If B is a system and $i \in \mathbb{Z}_n$ a valid thread index then we denote with $B_1(i)$ the behaviour executed in the thread i and with $B_2(i)$ its current status. Initially, all threads have status R, the current thread index is 0, and $B_1(i)$ is a behaviour expression of the shape f(v). It is a standard exercise to formalise a type system of simple first-order functional types for such a language and, in the following, we assume that all systems we consider are well typed.

Operational Semantics. The operational semantics is described by three relations of growing complexity, presented in Table 1: (1) $e \Downarrow v$, the closed expression e evaluates to the value v; (2) $(b, s) \xrightarrow{X} (b', s')$, the behaviour b with store s runs an atomic sequence of actions till b', producing a store s', and returning the control to the scheduler with status X; during an instant, we can have the following status transitions in a thread: $R \rightarrow S, W, N$ and $W \rightarrow R$, the last transition corresponds to a thread blocked on the behaviour match r with ... and no filters match the value of r; (3) $(B, s, i) \rightarrow (B', s', i')$ the system B with store s and current thread (index) i runs an atomic sequence of actions (performed by $B_1(i)$) and becomes (B', s', i').

Scheduler. The reduction relation, see Table 1, relies on the function \mathcal{N} that computes the index of the next thread that should run in the current instant and the function \mathcal{U} that updates the status of the thread at the end of an instant.

To ensure progress of the scheduling, we assume that if \mathcal{N} returns an index then it must be possible to run the corresponding thread in the current instant and that if \mathcal{N} is undefined (denoted $\mathcal{N}(...)\uparrow$) then no thread can be run in the current instant. In addition, one could arbitrarily enrich the functional behaviour of the scheduler by considering extensions such that \mathcal{N} depends on the history, the store, and/or is defined by means of probabilities. When no more thread can run, the instant ends and the following status transitions take place $N \to R$, $W \to R$. For simplicity, we assume here that every thread in status W takes the $[x] \Rightarrow \dots$ branch. Note that the function \mathcal{N} is undefined on the updated system if and only if all threads are stopped.

The Cooperative Fragment. The 'cooperative' fragment of the model with no synchrony is obtained by removing the next instruction and assuming that for all match instructions the branch $[x] \Rightarrow f(e)$ is such that f(...) = stop. Then all the interesting computation happens in the first instant, and in the second instant all the threads terminate. This fragment is already powerful enough to simulate, *e.g.*, Kahn networks (see examples in [1]).

Example 1 (Channels and Signals). As shown in our informal presentation of behaviours, the match instruction allows one to read a register subject to certain filter conditions. This is a powerful mechanism which recalls, *e.g.*, Linda communication [12], and that allows to encode various forms of channel and signal communication.

(1) We want to represent a *one place channel* c carrying values of type t. We introduce a new type ch(t) = empty | full of t and a register <math>c of type ch(t) with default value empty. A thread should send a message on c only if c is empty and it should receive a message only if c is *not* empty (a received message is discarded). These operations can be modelled using the following two derived operators:

send(c, e).b
$$=_{def}$$
 match c with empty \Rightarrow c := full(e).b
receive(c, x).b $=_{def}$ match c with full(x) \Rightarrow c := empty.b

(2) We want to represent a *fifo channel* c carrying values of type t such that a thread can always emit a value on c but may receive only if there is at least one message in the channel. We introduce a new type fch(t) = nil | cons of t * fch(t) and a register c of type fch(t) with default value nil. Hence a fifo channel is modelled by a register holding a list of values. We consider two read operations — freceive to fetch the first message on the channel and freceiveall to fetch the whole queue of messages — and we use the auxiliary function *insert* to queue messages at the end of the list:

$$\begin{array}{ll} \mbox{fsend}({\sf c},{\sf e}).b &=_{\rm def} \mbox{match } {\sf c} \mbox{ with } l \Rightarrow {\sf c} := insert(e,l).b \\ \mbox{freceive}({\sf c},x).b &=_{\rm def} \mbox{match } {\sf c} \mbox{ with } \mbox{cons}(x,l) \Rightarrow {\sf c} := l.b \\ \mbox{freceiveall}({\sf c},x).b &=_{\rm def} \mbox{match } {\sf c} \mbox{ with } \mbox{cons}(y,l) \Rightarrow {\sf c} := {\sf nil.}[\mbox{cons}(y,l)/x]b \\ \mbox{insert}(x,{\sf nil}) = {\sf cons}(x,{\sf nil}) \ , \qquad insert(x,{\sf cons}(y,l)) = {\sf cons}(y,insert(x,l)) \\ \end{array}$$

(3) We want to represent a signal s with the typical associated primitives: emitting a signal and blocking until a signal is present. We define a type sig =

73

Table 1	. 0	perational	semantics
---------	-----	------------	-----------

Expression evaluation:		
$\frac{e \Downarrow v}{c(e) \Downarrow c(v)} \qquad \frac{e \Downarrow v, f(p) = e, \sigma p = v, \sigma(e) \Downarrow v}{f(e) \Downarrow v}$		
Behaviour reduction:		
$(stop, s) \xrightarrow{S} (stop, s) \qquad (yield. b, s) \xrightarrow{R} (b, s) \qquad (next. f(e), s) \xrightarrow{N} (f(e), s)$		
$\frac{\text{no pattern matches } s(r)}{(match r with \dots, s)} \xrightarrow{W} (match r with \dots, s)$		
$\frac{\sigma p = s(r), (\sigma b, s) \xrightarrow{X} (b', s')}{(match r with \cdots \mid p \Rightarrow b \mid \dots, s) \xrightarrow{X} (b', s')}$		
$ \begin{array}{c c} \underline{e \Downarrow v, f(p) = b, \sigma p = v, (\sigma b, s) \xrightarrow{X} (b', s')} \\ \hline (f(e), s) \xrightarrow{X} (b', s') \end{array} \begin{array}{c} \underline{e \Downarrow v, (b, s[v/r]) \xrightarrow{X} (b', s')} \\ \hline (r := e.b, s) \xrightarrow{X} (b', s') \end{array} \end{array} $		
System reduction:		
$\frac{(B_1(i), s) \xrightarrow{X} (b', s'), B_2(i) = R, B' = B[(b', X)/i], \mathcal{N}(B', s', i) = k}{(B, s, i) \to (B'[(B'_1(k), R)/k], s', k)}$		
$\underbrace{\begin{array}{cccc} (B_1(i),s) \xrightarrow{X} (b',s'), & B_2(i) = R, & B' = B[(b',X)/i], & \mathcal{N}(B',s',i)\uparrow, \\ & & B'' = \mathcal{U}(B',s'), & \mathcal{N}(B'',s_o,0) = k \\ \hline & & & (B,s,i) \to (B'',s_o,k) \end{array}}$		
Conditions on the scheduler:		
If $\mathcal{N}(B, s, i) = j$ then $B_2(j) = R$ or $(B_2(j) = W$ and $B_1(j) = match r with \cdots \mid p \Rightarrow b \mid \ldots, \sigma p = s(r)$		
$ \begin{array}{ll} \text{If } \mathcal{N}(B,s,i) \uparrow \text{ then } & \forall k \in \mathbf{Z}_n, \ B_2(k) \in \{N,S\} \text{ or } (\ B_2(k) = W, \\ B_1(k) = \texttt{match r with } \dots \text{ and no pattern matches } s(r) \) \\ \mathcal{U}(B,s)(i) = \begin{cases} (b,S) & \text{if } B(i) = (b,S) \\ (b,R) & \text{if } B(i) = (b,N) \\ ([s(r)/x](f(e)),R) \text{ if } B(i) = (\texttt{match r with } \dots \mid [x] \Rightarrow f(e),W) \end{cases} $		

abst | prst and a register s of type *sig* with default value abst, meaning that a signal is originally absent:

 $\mathsf{emit}(\mathsf{s}).b =_{\mathsf{def}} \mathsf{s} := \mathsf{prst}.b$ wait(s).b =_{def} match s with $\mathsf{prst} \Rightarrow b$

3 Control Flow Analysis

To bound the resources needed for the execution of a system and make possible a compositional analysis, a preliminary control flow analysis is required. We require and statically check on the control flow, that threads can read any given register at most once in an instant. The following simple example shows that *without* the read once restriction, a thread can use a register as an accumulator and produce an exponential growth of the size of the data within an instant.

Example 2. Let $nat = z \mid s \text{ of } nat$ be the type of tally natural numbers. The function *dble*, defined by the two rules dble(z) = z and dble(s(n)) = s(s(dble(n))) doubles a number so that |dble(n)| = 2|n|. We assume r is a register of type *nat* with initial value s(z). Now consider the following recursive behaviour:

exp(z) = stop, $exp(s(n)) = match r with m \Rightarrow r := dble(m).exp(n)$

The evaluation of exp(n) involves |n| reads to the register r and, after each read operation, the size of the value stored in r doubles. Hence, at end of the instant, the register contains a value of size $2^{|n|}$.

The read once condition is comparable to the restriction on the absence of immediate cyclic definitions in LUSTRE and does not appear to be a severe limitation on the expressiveness of the language. An important consequence of the *read once* condition is that a behaviour can be described as a *function* of its parameters and the registers it may read during an instant. We stress that we retain the *read once* condition for its simplicity, however it is clear that one could weaken the condition and adapt the analysis given in Section 3.1 to allow the execution of a read instruction at most a constant number of times.

3.1 Enforcing the Read Once Condition

We now describe a simple analysis that guarantees the read once condition. Consider the set $Reg = \{r_1, \ldots, r_m\}$ of the registers as an alphabet. To every function symbol f whose result is a behaviour, we associate the least language R(f) of words over Reg such that ϵ , the empty word, is in R(f) and the following conditions are satisfied:

$$\begin{aligned} &\text{if } (f(\boldsymbol{p}_i) = b_i)_{i \in 1..n} \text{ are the rules of } f \text{ then } R(f) =_{\text{def}} R(f) \cdot \bigcup_{i \in 1..n} R(b_i) , \\ &R(\text{match } \mathsf{r} \text{ with } p_1 \Rightarrow b_1 \mid \dots \mid p_n \Rightarrow b_n \mid [x] \Rightarrow g(\boldsymbol{e})) =_{\text{def}} \{\mathsf{r}\} \cdot \bigcup_{i \in 1..n} R(b_i) , \\ &R(\text{stop}) = \{\epsilon\} , \qquad R(g(\boldsymbol{e})) = R(g) , \qquad R(\mathsf{r} := e.b) = R(b) , \\ &R(\text{yield}.b) = R(b) , \qquad R(\text{next.} g(\boldsymbol{e})) = \{\epsilon\} . \end{aligned}$$

Looking at the words in R(f), we get an over-approximation of the sequences of registers that a thread can read in an instant starting from the control point

f with arbitrary parameters and store. Note that an expression can never read or write a register.

To determine the sets R(f), we perform an iterative computation according to the equations above. The iteration stops when either (1) we reach a fixpoint (and we are sure that the property holds) or (2) we notice that a word in the current approximation of R(f) contains the same register twice (thus we never need to consider words whose length is greater than the number of registers). If the first situation occurs, then for every function symbol f that returns a behaviour we can obtain a list of registers \mathbf{r}_f that a thread starting from control point f may read. We are going to consider these registers as *hidden parameters* (variables) of the function f. If the second condition occurs, we cannot guarantee the read once property and we stop analysing the code.

Example 3. This will be the running example for this section. We consider the representation of signals as in Example 1(3). We assume two signals sig and ring. The behaviour alarm(n, m) will emit a signal on ring if it detects that no signal is emitted on sig for m consecutive instants. The alarm delay is reset to n if the signal sig is present.

alarm(x, z) = ring := prst.stop, $alarm(x, s(y)) = match sig with prst \Rightarrow next. <math>alarm(x, x) | [_] \Rightarrow alarm(x, y)$

By computing R on this example, we obtain: $R(alarm) = \{\epsilon\} \cdot (R(ring := prst.stop) \cup R(match sig with ...)) = \{\epsilon\} \cdot (\{\epsilon\} \cup (\{sig\} \cdot \{\epsilon\})) = \{\epsilon, sig\}.$

3.2 Control Points

We define a symbolic representation of the set of states reachable by a thread based on the control flow graph of its behaviours. A *control point* is a triple (f(p), be, i) where, intuitively, f is the currently called function, p represents the patterns crossed so far in the function definition plus possibly the registers that still have to be read, *be* is the continuation, and *i* is an integer flag in $\{0,1,2\}$ that will be used to associate with the control point various kinds of conditions. We associate with a system satisfying the read once condition a *finite* number of control points. If the function f returns a value and is defined by the rules $f(p_1) = e_1, \ldots, f(p_n) = e_n$, then we associate with f the set $\{(f(p_1), e_1, 0), \ldots, (f(p_n), e_n, 0)\}$.

On the other hand, if the function f is a behaviour defined by the rules $f(p_1) = b_1, \ldots, f(p_n) = b_n$ then the computation of the control points proceeds as follows. We assume that the registers have been ordered and that for every behaviour definition f, we have an ordered vector \mathbf{r}_f of registers that may be read within an instant starting from f. (The vector \mathbf{r}_f is obtained from R(f)). With every such f we associate a fresh function symbol f^+ whose arity is that of f plus the length of \mathbf{r}_f and we regard the registers as part of the formal parameters of f^+ . Then from the definition of f we produce the set $\bigcup_{i \in 1...n} C(f^+, (p_i, \mathbf{r}_f), b_i)$, where $C(f^+, \mathbf{p}, \mathbf{b})$ is defined inductively on \mathbf{b} as follows:

$$\begin{split} \mathcal{C}(f^+\!,p,b) &= \operatorname{case} b \text{ of} \\ \text{stop} &: \{(f^+(p),b,2)\} \\ g(e) &: \{(f^+(p),b,0)\} \\ \text{yield.}b' &: \{(f^+(p),b,2)\} \cup \mathcal{C}(f^+\!,p,b') \\ \text{next.}g(e) &: \{(f^+(p),b,2), (f^+(p),g(e),2)\} \\ \text{r} &:= e.b' &: \{(f^+(p),b,2), (f^+(p),e,1)\} \cup \mathcal{C}(f^+\!,p,b') \\ \text{match r with } p_1 \Rightarrow b_1 \mid \dots \mid p_n \Rightarrow b_n \mid [x] \Rightarrow g(e) :: \{(f^+(p),b,2), (f^+([p_1/r]p),b_1) \cup \dots \cup \mathcal{C}(f^+,([p_n/r]p),b_n) \\ \end{split}$$

By inspecting the definitions, we can check that a control point $(f(\mathbf{p}), be, i)$ has the property that $Var(be) \subseteq Var(\mathbf{p})$. The read once condition is instrumental to this property. For instance, (i) in case g(e), we know that if g can read some register r then r could not have been already read by f and (ii) in the case of the match operator, we know that the register r has not been already read by f. Hence, in these two cases, the register r must still occur in p.

Example 4. With reference to Example 3, we obtain the following control points:

 $\begin{array}{ll} (alarm^+(x, \mathsf{z}, \mathsf{sig}), \mathsf{ring} := \mathsf{prst.stop}, 2) & (alarm^+(x, \mathsf{z}, \mathsf{sig}), \mathsf{prst}, 1) \\ (alarm^+(x, \mathsf{z}, \mathsf{sig}), \mathsf{stop}, 2) & (alarm^+(x, \mathsf{s}(y), \mathsf{sig}), \mathsf{match} \ldots, 2) \\ (alarm^+(x, \mathsf{s}(y), \mathsf{prst}), \mathsf{next.}alarm(x, x), 2) & (alarm^+(x, \mathsf{s}(y), \mathsf{prst}), alarm(x, x), 2) \\ (alarm^+(x, \mathsf{s}(y), _), alarm(x, y), 2) \end{array}$

Definition 1. An instance of a control point $(f(\mathbf{p}), \mathbf{b}, \mathbf{i})$ is a behaviour $\mathbf{b}' = \sigma \mathbf{b}$, where σ is a substitution mapping the free variables in \mathbf{b} to values.

The property of being an instance of a control point is preserved by (behaviour and) system reduction. Thus the control points associated with a system do provide a representation of all reachable configurations.

Proposition 1. Suppose $(B, s, i) \rightarrow (B', s', i')$ and that for all thread indexes $j \in \mathbb{Z}_n$, $B_1(j)$ is an instance of a control point. Then for all $j \in \mathbb{Z}_n$, we have that $B'_1(j)$ is an instance of a control point.

In order to prove the termination of the instant and to obtain a bound on the size of computed value, we associate order constraints to control points as follows:

Control point: $(f(p), e, 0), (f^+(p), g(e), 0), (f^+(p), e, 1), (f^+(p), be, 2)$ Constraint: $f(p) \succ_0 e, f^+(p) \succ_0 g^+(e, \mathsf{r}_g), f^+(p) \succ_1 e, no constraints$

We say that a constraint $e \succ_i e'$ has index *i*. We rely on the constraints of index 0 to enforce termination of the instant and on those of index 0 or 1 to enforce a bound on the size of the computed values. Note that the constraints are on pure first order terms, a property that allows us to reuse techniques developed in the standard term rewriting framework.

77

Example 5. With reference to the control points in Example 4, we obtain the constraint $alarm^+(x, z, sig) \succ_1 prst$. We note that no constraints of index 0 are generated and so in this simple case the control flow analysis can already establish the termination of the thread and all is left to do is to check that the size of the data is under control, which will also be easily verified.

4 Termination of the Instant

We recall that a *reduction order* > over first-order terms is a well-founded order that is closed under context and substitution: t > s implies C[t] > C[s] and $\sigma t > \sigma s$, where C is any one hole context and σ is any substitution (see, e.g., [10]).

Definition 2 (Termination Condition). We say that a system satisfies the termination condition if there is a reduction order > such that all constraints of index 0 associated with the system hold in the reduction order.

In this section, we assume that the system satisfies the termination condition. As expected this entails that the evaluation of closed expressions succeeds.

Proposition 2. Let e be a closed expression. Then there is a value v such that $e \Downarrow v$ and $e \ge v$ with respect to the reduction order.

Moreover, the following proposition states that a behaviour will always return the control to the scheduler.

Proposition 3 (Progress). Let b be an instance of a control point. Then for all stores s, $(b, s) \xrightarrow{X} (b', s')$.

Finally, we show that at each instant the system will reach a configuration in which the scheduler detects the end of the instant and proceeds to the reinitialisation of the store and the status (as specified by rule (s_2) in Table 1).

Theorem 1 (Termination of the Instant). All sequences of system reductions involving only rule (s_1) are finite.

Proposition 3 and Theorem 1 are proven by exhibiting a suitable well-founded measure which is based both on the reduction order and the fact that the number of reads a thread may perform in an instant is finite.

Example 6. We consider a recursive behaviour monitoring the register i (acting as a fifo channel) and parameterised on a number x representing the largest value read so far. At each instant, the behaviour reads the list l of values received on i and assigns to o the greatest number in x and l.

$$f(x) = \text{yield.match i with } l \Rightarrow f_1(maxl(l, x)) \qquad f_1(x) = o := x.\text{next.} f(x)$$

 $\begin{array}{ll} \max(\mathsf{z},y) = y \ , & \max(\mathsf{s}(x),\mathsf{z}) = \mathsf{s}(x) \ , & \max(\mathsf{s}(x),\mathsf{s}(y)) = \mathsf{s}(\max(x,y)) \\ \max(\mathsf{nil},x) = x \ , & \max(\mathsf{cons}(y,l),x) = \max(l,\max(y,x)) \end{array}$

It is easy to prove the termination of the thread by recursive path ordering, where the function symbols are ordered as $f^+ > f_1^+ > maxl > max$, the arguments of *maxl* are compared lexicographically from left to right, and the constructor symbols are incomparable and smaller than any function symbol.

5 Quasi-Interpretations

Our next task is to control the size of the values computed by the threads. A suitable notion of quasi-interpretation [17,3] provides a modular solution to this problem.

Definition 3 (Assignment). *Given a program, an* assignment q associates with constructors and function symbols, functions over the positive reals \mathbb{R}^+ such that:

(1) If c is a constant then q_c is the constant 0,

(2) If **c** is a constructor with arity $n \ge 1$ then q_c is the function in $(\mathbb{R}^+)^n \to \mathbb{R}^+$ such that $q_c(x_1, \ldots, x_n) = d + \sum_{i \in 1 \ldots n} x_i$, for some $d \ge 1$,

(3) if f is a function (identifier) with arity n then $q_f : (\mathbb{R}^+)^n \to \mathbb{R}^+$ is monotonic and for all $i \in 1..n$ we have $q_f(x_1, \ldots, x_n) \ge x_i$.

An assignment q is extended to all expressions e as follows, giving a function expression q_e with variables in Var(e):

$$q_x = x$$
, $q_{c(e_1,\ldots,e_n)} = q_c(q_{e_1},\ldots,q_{e_n})$, $q_{f(e_1,\ldots,e_n)} = q_f(q_{e_1},\ldots,q_{e_n})$.

It is easy to check that for all values v, there exists a constant d depending on the quasi-interpretation such that: $|v| \le q_v \le d \cdot |v|$.

Definition 4 (Quasi-Interpretation). An assignment is a quasi-interpretation, if for all constraints associated with the system of the shape $f(\mathbf{p}) \succ_i e$, with $i \in \{0,1\}$, the inequality $q_{f(\mathbf{p})} \geq q_e$ holds over the non-negative reals.

Quasi-interpretations are designed so as to provide a bound on the size of the computed values as a function of the size of the input data. In the following, we assume given a suitable quasi-interpretation, q, for the system under investigation.

Example 7. With reference to Examples 2 and 6, the following assignment is a quasi-interpretation (we give no quasi-interpretations for the function *exp* because it fails the read once condition):

$$\begin{array}{l} q_{\mathsf{nil}} = q_{\mathsf{z}} = 0 \;, \quad q_{\mathsf{s}}(x) = x + 1 \;, \quad q_{\mathsf{cons}}(x,l) = x + l + 1 \;, \quad q_{dble}(x) = 2 \cdot x \;, \\ q_{f^+}(x,\mathsf{i}) = x + \mathsf{i} + 1 \;, \quad q_{f^+_1}(x) = x \;, \quad q_{maxl}(x,y) = q_{max}(x,y) = max(x,y) \;. \end{array}$$

One can show [3] that in the purely functional fragment of our language every value v computed during the evaluation of an expression $f(v_1, \ldots, v_n)$ satisfies the following condition:

$$|v| \leq q_v \leq q_{f(v_1,\ldots,v_n)} = q_f(q_{v_1},\ldots,q_{v_n}) \leq q_f(d|v_1|,\ldots,d|v_n|) .$$
(1)

We generalise this result to threads as follows.

79

Theorem 2. Given a system of synchronous threads B, suppose that at the beginning of the instant $B_1(i) = f(v)$ for some thread index i. Then the size of the values computed by the thread i during an instant is bound by $q_{f^+(v,u)}$ where u are the values contained in the registers \mathbf{r}_f when they are read by the thread (or some constant value, otherwise).

Theorem 2 is proven by showing that quasi-interpretations satisfy a suitable invariant. In general, a value computed and written by a thread can be read by another thread. However, at each instant, we have a bound on the number of threads and the number of reads that can be performed. We can then derive a bound on the size of the computed values which depends only on the size of the parameters at the beginning of the instant.

Corollary 1. Let B be a system with **m** registers and **n** threads. Suppose $B_1(i) = f_i(v_i)$ for $i \in \mathbf{Z}_n$. Let **c** be a bound of the size of the largest parameter of the functions f_i and the largest default value of the registers. Suppose **h** is a function bounding all the quasi-interpretations, that is, for all the functions f_i^+ we have $h(x) \ge q_{f_i^+}(x, \ldots, x)$ over the non-negative reals. Then the size of the values computed by the system B during an instant is bound by $h^{n\cdot m+1}(c)$.

Example 8. The $n \cdot m$ iterations of the function h predicted by Corollary 1 correspond to a tight bound, as shown by the following example. We assume n threads and m registers (with default value z). The control of each thread is described as follows, where *writeall(e).b* stands for the behaviour $r_1 := e \cdots r_m := e.b$:

$$f(x_0) = \operatorname{match} r_1 \operatorname{with} x_1 \Rightarrow \operatorname{writeall}(dble(max(x_1, x_0))).$$

match $r_2 \operatorname{with} x_2 \Rightarrow \operatorname{writeall}(dble(x_2)).$
.....

match \mathbf{r}_m with $x_m \Rightarrow writeall(dble(x_m)).next.f(dble(x_m))$.

For this system we have $c \ge |x_0|$ and $h(x) = q_{dble}(x) = 2 \cdot x$. It is easy to show that, at the end of an instant, there have been $m \cdot n$ assignments to each register (*m* for every thread in the system) and that the value stored in each register is $dble^{m \cdot n}(x_0)$ of size $2^{m \cdot n} \cdot |x_0|$.

6 Combining Termination and Quasi-interpretations

To bound the space needed for the execution of a system during an instant we also need to bound the number of nested recursive calls, *i.e.*, the number of frames that can be found on the stack (a precise definition of frame is given in the long version of this paper [1]). Unfortunately, quasi-interpretations provide a bound on the size of the frames but not on their number (at least not in a direct implementation that does not rely on memoization). One way to cope with this problem is to combine quasi-interpretations with various families of reduction orders [9,17]. In the following, we provide an example of this approach based on *recursive path orders* which is a widely used and fully mechanisable technique to prove termination [10].

Definition 5. We say that a system terminates by LPO, if the reduction order associated with the system is a recursive path order where: (1) function symbols are compared lexicographically; (2) constructor symbols are always smaller than function symbols and two distinct constructor symbols are incomparable; (3) the arguments of constructor symbols are compared componentwise (product order).

Definition 6. We say that a system admits a polynomial quasi-interpretation if it has a quasi-interpretation where all functions are bound by a polynomial.

Theorem 3. If a system B terminates by LPO and admits a polynomial quasiinterpretation then the computation of the system in an instant runs in space polynomial in the size of the parameters of the threads at the beginning of the instant.

The proof of Theorem 3 is based on Corollary 1 that provides a polynomial bound on the size of the computed values and on an analysis of nested calls in the LPO order that can be found in [9]. The point is that the depth of such nested calls is polynomial in the size of the values, which allows us to effectively compute a polynomial bounding the space necessary for the execution of the system. We stress that beyond proving that a system 'runs in PSPACE', we can extract a definite polynomial that depends on the quasi-interpretation and that bounds the size needed to run a system during an instant.

Example 9. With reference to Example 6, we can check that the order used there is indeed a LPO. From the quasi-interpretation in Example 7, we can deduce that the function h(x) has the shape $a \cdot x + b$ (it is affine). More precisely, we can choose $h(x) = 2 \cdot x + 1$. In practice, many useful functions admit quasi-interpretations bound by an affine function such as the max-plus polynomials considered in [3]. Note that the parameter of the thread is the largest value received so far. Clearly, bounding the value of this parameter for arbitrary many instants requires a global analysis of the system.

7 Conclusion

The execution of a thread in a cooperative synchronous model can be regarded as a sequence of instants. One can make each instant simple enough so that it can be described as a function — our experiments with writing sample programs show that the restrictions we impose do not hinder the expressivity of the language. Then well-known static analyses used to bound the resources needed for the execution of first-order functional programs can be extended to handle systems of synchronous cooperative threads. We believe this provides some evidence for the relevance of these techniques in concurrent/embedded programming. We also expect that our approach can be extended to a richer programming model including, *e.g.*, references as first-class values, transactions-like primitives for error recovery, more elaborate mechanisms for preemption, …

The static analyses we have considered do not try to analyse the whole system. On the contrary, they focus on each thread separately and can be carried

81

out incrementally. On the basis of our previous work [2] and the virtual machine presented in [1], we expect that these analyses can be performed at bytecode level. These characteristics are particularly interesting in the framework of 'mobile code' where threads can enter or leave the system at the end of each instant as described in [5].

References

- 1. R. Amadio and S. Dal-Zilio. Resource control for synchronous cooperative threads. Research Report LIF 22-2004, 2004.
- R. Amadio, S. Coupet-Grimal, S. Dal-Zilio, and L. Jakubiec. A functional scenario for bytecode verification of resource bounds. Research Report LIF 17-2004, 2004.
- 3. R. Amadio. Max-plus quasi-interpretations. In Proc. *TLCA*, Springer LNCS 2701, 2003.
- 4. J. Armstrong, R. Virding, C. Wikström, M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall 1996.
- 5. G. Boudol, ULM, a core programming model for global computing. In Proc. *ESOP*, Springer LNCS 2986, 2004.
- 6. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
- 7. F. Boussinot and R. De Simone, The SL Synchronous Language. *IEEE Trans. on* Software Engineering, 22(4):256–266, 1996.
- 8. G. Berry and G. Gonthier, The Esterel synchronous programming language. *Science of computer programming*, 19(2):87–152, 1992.
- 9. G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On termination methods with space bound certifications. In Proc. *PSI*, Springer LNCS 2244, 2001.
- 10. F. Baader and T. Nipkow. Term rewriting and all that. CUP, 1998.
- 11. P. Baillot and V. Mogbil, Soft lambda calculus: a language for polynomial time computation. In Proc. *FoSSaCS*, Springer LNCS 2987, 2004.
- 12. N. Carriero and D. Gelernter. Linda in Context. CACM, 32(4): 444-458, 1989.
- 13. A. Cobham. The intrinsic computational difficulty of functions. In Proc. Logic, *Methodology, and Philosophy of Science II,* North Holland, 1965.
- 14. M. Hofmann. The strength of non size-increasing computation. In Proc. *POPL*, ACM Press, 2002.
- 15. N. Jones. *Computability and complexity, from a programming perspective*. MIT-Press, 1997.
- D. Leivant. Predicative recurrence and computational complexity i: word recurrence and poly-time. *Feasible mathematics II, Clote and Remmel (eds.)*, Birkhäuser:320–343, 1994.
- 17. J.-Y. Marion. *Complexité implicite des calculs, de la théorie à la pratique*. Habilitation à diriger des recherches, Université de Nancy, 2000.
- 18. M. Odersky. Functional nets. In Proc. ESOP, Springer LNCS 1782, 2000.
- 19. J. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at the USENIX Technical Conference, 1996.
- P. Puschner and A. Burns (eds.), *Real time systems* 18(2/3), special issue on Worst-Case Execution Time Analysis, 2000.

Verifying Finite-State Graph Grammars: An Unfolding-Based Approach*

Paolo Baldan¹, Andrea Corradini², and Barbara König³

¹ Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy baldan@dsi.unive.it

² Dipartimento di Informatica, Università di Pisa, Italy

andrea@di.unipi.it

³ Institut für Formale Methoden der Informatik, Universität Stuttgart, Germany koenigba@fmi.uni-stuttgart.de

Abstract. We propose a framework where behavioural properties of finite-state systems modelled as graph transformation systems can be expressed and verified. The technique is based on the unfolding semantics and it generalises McMillan's complete prefix approach, originally developed for Petri nets, to graph transformation systems. It allows to check properties of the graphs reachable in the system, expressed in a monadic second order logic.

1 Introduction

Graph transformation systems (GTSs) are recognised as an expressive specification formalism, properly generalising Petri nets and especially suited for concurrent and distributed systems [9]: the (topo)logical distribution of a system can be naturally represented by using a graphical structure and the dynamics of the system, e.g., the reconfigurations of its topology, can be modelled by means of graph rewriting rules.

The concurrent behaviour of GTSs has been thoroughly studied and a consolidated theory of concurrency for GTSs is available, including the generalisation of several semantics of Petri nets, like process and unfolding semantics (see, e.g., [6, 20, 3]). However, only recently, building on these semantical foundations, some efforts have been devoted to the development of frameworks where behavioural properties of GTSs can be expressed and verified (see [12, 15, 13, 21, 19, 1]).

As witnessed, e.g., by the approaches in [17, 10] for Petri Nets, truly concurrent semantics are potentially useful in the verification of finite-state systems, in that they help to avoid the combinatorial explosion arising when one explores all possible interleavings of events. Still, to the best of our knowledge, no technique based on partial order (process or unfolding) semantics has been proposed for the verification of finite-state GTSs.

^{*} Research partially supported by EU FET-GC Project AGILE, the EC RTN SEGRA VIS, DFG project SANDS and EPSRC grant R93346/01.

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 83-98, 2004.

[©] Springer-Verlag Berlin Heidelberg 2004

In this paper we contribute to this topic by proposing a verification framework for *finite-state graph transformation systems* based on their unfolding semantics. Our technique is inspired by the approach originally developed by McMillan for Petri nets [17] and further developed by many authors (see, e.g., [10,11,23]). More precisely, our technique applies to any *graph grammar*, i.e., any set of graph rewriting rules with a fixed start graph (the initial state of the system), which is *finite-state* in a liberal sense: the set of graphs which can be reached from the start graph, considered not only *up to isomorphism*, but also *up to isolated nodes*, is finite. Hence in a finite-state graph grammar in our sense there is not actually a bound to the number of nodes generated in a computation, but only to the nodes which are connected to some edge at each stage of the computation. Existing model-checking tools, such as SPIN [14], usually do not directly support the creation of an arbitrary number of objects while still maintaining a finite state space, making entirely non-trivial their use for checking finite-state GTSs (similar problems arise for process calculi agents with name creation).

As a first step we face the problem of identifying a finite, still useful fragment of the unfolding of a GTS. In fact, the unfolding construction for GTSs produces a structure which fully describes the concurrent behaviour of the system, including all possible steps and their mutual dependencies, as well as all reachable states. However, the unfolding is infinite for non-trivial systems, and cannot be used directly for model-checking purposes.

Following McMillan's approach, we show that given any finite-state graph grammar \mathcal{G} a *finite* fragment of its unfolding which is *complete*, i.e., which provides full information about the system as far as reachability (and other) properties are concerned, can be characterised as the maximal prefix of the unfolding not including *cut-off events*. The greater expressiveness of GTSs, and specifically, the possibility of performing "contextual" rewritings (i.e., of preserving part of the state in a rewriting step), a feature which leads to multiple local histories for a single event (see, e.g., the work on contextual nets [18, 22, 4, 23]), imposes a generalisation of the original notion of cut-off.

Unfortunately the characterisation of the finite complete prefix is not constructive. Hence, while leaving as an open problem the definition of a general algorithm for constructing such a prefix, we identify a significant subclass of graph grammars where an adaptation of the existing algorithms for Petri nets is feasible. These are called *read-persistent* graph grammars by analogy with the terminology used in the work on contextual nets [23].

In the second part we consider a logic $\mathcal{L}2$ where graph properties of interest can be expressed, like the non-existence and non-adjacency of edges with specific labels, the absence of certain paths (related to security properties) or cycles (related to deadlock-freedom). This is a monadic second-order logic over graphs where quantification is allowed over (sets of) edges. (Similar logics are considered in [8] and, in the field of verification, in [19, 5].) Then we show how a complete finite prefix of a grammar \mathcal{G} can be used to verify properties, expressed in $\mathcal{L}2$, of the graphs reachable in \mathcal{G} . This is done by exploiting both the graphical structure underlying the prefix and the concurrency information it provides.

The rest of the paper is organised as follows. Section 2 introduces graph transformation systems and their unfolding semantics. Section 3 studies finite complete prefixes for finite-state GTSs. Section 4 introduces a logic for GTSs, showing how it can be checked over a finite complete prefix. Finally, Section 5 draws some conclusions and indicates directions of further research. A more detailed presentation of the material in this paper can be found in [2].

2 Unfolding Semantics of Graph Grammars

This section presents the notion of graph rewriting used in the paper. Rewriting takes place on so-called *typed graphs*, namely graphs labelled over a structure that is itself a graph [6]. It can be seen as a set-theoretical presentation of an instance of algebraic (single- or double-pushout) rewriting (see, e.g., [7]). Next we review the notion of occurrence grammar, which is instrumental in defining the unfolding of a graph grammar [3, 20].

2.1 Graph Transformation Systems

In the following, given a set A we denote by A^* the set of finite strings of elements of A. Given $u \in A^*$ we write |u| to indicate the length of u. If $u = a_0 \dots a_n$ and $0 \le i \le n$, by $[u]_i$ we denote the i-th element a_i of u. Furthermore, if $f : A \to B$ is a function then we denote by $f^* : A^* \to B^*$ its extension to strings.

A (hyper)graph G is a tuple (V_G, E_G, c_G) , where V_G is a set of nodes, E_G is a set of edges and $c_G : E_G \to V_G^*$ is a connection function. A node $v \in V_G$ is called *isolated* if it is not connected to any edge. Given two graphs G, G', a graph morphism $\phi : G \to G'$ is a pair $\langle \phi_V : V_G \to V_{G'}, \phi_E : E_G \to E_{G'} \rangle$ of total functions such that for all $e \in E_G, \phi_V^*(c_G(e)) = c_{G'}(\phi_E(e))$. When obvious from the context, the subscripts V and E will be omitted.

Definition 1 (Typed Graph). Given a graph (of types) T, a typed graph G over T is a graph |G|, together with a morphism $type_G : |G| \to T$. A morphism between T-typed graphs $f : G_1 \to G_2$ is a graph morphism $f : |G_1| \to |G_2|$ consistent with the typing, i.e., such that $type_{G_1} = type_{G_2} \circ f$.

A typed graph G is called *injective* if the typing morphism $type_G$ is injective. More generally, given $n \in \mathbb{N}$, the graph is called *n*-*injective* if for any item x in T, $|type_G^{-1}(x)| \leq n$, namely if the number of "instances of resources" of any type x is bounded by n. Given two (typed) graphs G and G' we will write $G \simeq G'$ to mean that G and G' are *isomorphic*, and $G \simeq G'$ when G and G' are *isomorphic* up to *isolated nodes*, i.e., once their isolated nodes have been removed.

In the sequel we extensively use the fact that given a graph G, any subgraph of G without isolated nodes is identified by the set of its edges. Precisely, given a subset of edges $X \subseteq E_G$, we denote by graph(X) the least subgraph of G (actually the unique subgraph, up to isolated nodes) having X as set of edges.

We will use some set-theoretical operations on (typed) graphs with "componentwise" meaning. Let G and G' be T-typed graphs. We say that G and G'

are *consistent* if $G \cup G'$ defined as $(V_{|G|} \cup V_{|G'|}, E_{|G|} \cup E_{|G'|}, c_G \cup c_{G'})$, typed by $type_G \cup type_{G'}$, is a well-defined *T*-typed graph. In this case also the intersection $G \cap G'$, constructed in a similar way, is well-defined. Given a graph *G* and a set (of edges) *E* we denote by G - E the graph obtained from *G* by removing the edges in *E*. Sometimes we will also refer to the items (nodes and edges) in G - G', where *G* and *G'* are graphs, although the structure resulting as the componentwise set-difference of *G* and *G'* might not be a well-defined graph.

Definition 2 (Production). Given a graph of types T, a T-typed production is a pair of finite consistent T-typed graphs q = (L, R), often written $L \to R$, such that 1) $L \cup R$ and L do not include isolated nodes; 2) $V_{|L|} \subseteq V_{|R|}$; and 3) $E_{|L|} - E_{|R|}$ and $E_{|R|} - E_{|L|}$ are non-empty.

A rule $L \to R$ specifies that, once an occurrence of L is found in a graph G, then G can be rewritten by removing (the images in G of) the items in L - R and adding those in R - L. The (images in G of the) items in $L \cap R$ instead are left unchanged: they are, in a sense, preserved or read by the rewriting step.

This informal explanation should also motivate Conditions 1–3 above. Condition 1 essentially states that we are interested only in rewriting up to isolated nodes: by the requirement on $L \cup R$, no node is isolated when created and, by the requirement on L, nodes that become isolated have no influence on further reductions. Thus one can safely assume that isolated nodes are removed by some kind of garbage collection. Consistently with this view, by Condition 2 productions cannot delete nodes (deletion can be simulated by leaving that node isolated). Condition 3 ensures that every production consumes and produces at least one edge: a requirement corresponding to T-restrictedness in Petri net theory.

Definition 3 (Graph Rewriting). Let $q = L \rightarrow R$ be a *T*-typed production. A match of q in a *T*-typed graph G is a morphism $\phi : L \rightarrow G$, satisfying the identification condition, i.e., for $e, e' \in E_{|L|}$, if $\phi(e) = \phi(e')$ then $e, e' \in E_{|R|}$. In this case G rewrites to the graph H, obtained as $H = ((G - \phi(E_{|L|} - E_{|R|})) \uplus R)/=$, where \equiv is the least equivalence on the items of the graph such that $x \equiv \phi(x)$. We write $G \Rightarrow_{q,\phi} H$ or simply $G \Rightarrow_q H$.

A rewriting step is schematically represented in Fig. 1. Intuitively, in the graph $H' = G - \phi(E_{|L|} - E_{|R|})$ the images of all the edges in L - R have been removed. Then in order to get the resulting graph, merge R to H' along the image through ϕ of the preserved subgraph $L \cap R$. Formally the resulting graph H is obtained by first taking $H' \oplus R$ and then by identifying, via the equivalence \equiv , the image through ϕ of each item in $L \cap R$ with the corresponding item in R.

Definition 4 (Graph Transformation System and Graph Grammar). A graph transformation system (GTS) is a triple $\mathcal{R} = \langle T, P, \pi \rangle$, where T is a graph of types, P is a set of production names and π is a function mapping each production name $q \in P$ to a T-typed production $\pi(q) = L_q \rightarrow R_q$. A graph grammar is a tuple $\mathcal{G} = \langle T, G_s, P, \pi \rangle$ where $\langle T, P, \pi \rangle$ is a GTS and G_s is a finite T-typed graph, without isolated nodes, called the start graph. We denote



Fig. 1. A rewriting step, schematically

by $Elem(\mathcal{G})$ the (disjoint) union $E_T \uplus P$, i.e., the set of edges in the graph of types and the production names. We call \mathcal{G} finite if the set $Elem(\mathcal{G})$ is finite.

A *T*-typed graph *G* is *reachable* in \mathcal{G} if $G_s \Rightarrow_{\mathcal{G}}^* G'$ for some $G' \simeq G$, where $\Rightarrow_{\mathcal{G}}^*$ is the transitive closure of the rewriting relation induced by productions in \mathcal{G} .

We remark that Place/Transition Petri nets can be viewed as a special subclass of typed graph grammars. Say that a graph G is *edge-discrete* if its set of nodes is empty (and thus edges have no connections). Given a P/T net P, let T_P be the edge-discrete graph having the set of places of P as edges. Then any finite edge-discrete graph typed over T_P can be seen as a marking of P: an edge typed over s represents a token in place s. Using this correspondence, a production $L_t \rightarrow R_t$ faithfully represents a transition t of P if L_t encodes the marking *pre-set*(t), R_t encodes *post-set*(t), and $L_t \cap R_t = \emptyset$. The graph grammar corresponding to a Petri net is finite iff the original net has finitely many places and transitions. Observe that the generalisation from edge-discrete to proper graphs radically changes the expressive power of the formalism. For instance, unlike P/T Petri nets, the class of grammars in this paper is Turing complete.

Example 1. Consider the graph grammar CP, modeling a system where three *processes* of type P are connected to a *communication manager* of type CM (see the start graph in Fig. 2, where edges are represented as rectangles and nodes as small circles). Two processes may establish a new connection with each other via the communication manager, becoming *processes engaged* in communication (typed *PE*, the only edge with more than one connection). This transformation is modelled by the production [engage] in Fig. 2: observe that a new node connecting the two processes is created. The second production [release] terminates the communication between two partners. A typed graph G over T_{CP} is drawn by labeling each edge or node x of G with ": $type_G(x)$ ". Only when the same graphical item x belongs to both the left- and the right-hand side of a production we include its identity in the label (which becomes " $x : type_G(e)$ "): in this case we also shade the item, to stress that it is preserved by the production.

The notion of safety for graph grammars [6] generalises the one for P/T nets which requires that each place contains at most one token in any reachable marking. More generally, we extend to graph grammars the notion of *n*-boundedness.



Fig. 2. The finite-state graph grammar \mathcal{CP}

Definition 5 (Bounded/Safe Grammar). For a fixed $n \in \mathbb{N}$, we say that a graph grammar \mathcal{G} is *n*-bounded if for all graphs H reachable in \mathcal{G} there is an *n*-injective graph H' such that $H' \cong H$. A 1-bounded grammar will be called safe.

The definition can be understood by thinking of edges of the graph of types T as a generalisation of places in Petri nets. In this view the number of different edges of a graph which are typed on the same item of T corresponds to the number of tokens contained in a place. Observe that for *finite* graph grammars, *n*-boundedness amounts to the property of being finite-state (up to isomorphism and up to isolated nodes). In the sequel when considering a finite-state graph grammar we will (often implicitly) assume that it is also finite.

For instance, the graph grammar CP in Fig. 2 is clearly 3-bounded and thus finite-state (but only up to isolated nodes).

2.2 Nondeterministic Occurrence Grammars

When a graph grammar \mathcal{G} is safe, and thus reachable graphs are injectively typed, at every step, for any item t in the type graph every production can consume, preserve and produce a single item typed t. Hence we can safely think that a production, according to its typing, *consumes, preserves* and *produces* items of the graph of types. Using a net-like language, we speak of *pre-set* \P , *context* \underline{q} and *post-set* q^{\bullet} of a production q. Since we work with graphs considered up to isolated nodes, we will record in these sets only edges. Formally, for any production q of a graph grammar $\mathcal{G} = \langle T, G_s, P, \pi \rangle$, we define

 $\begin{array}{ll} ^{\bullet}q = type_{L_q}(E_{|L_q|} - E_{|R_q|}) & \underline{q} = type_{L_q}(E_{|L_q \cap R_q|}) & q^{\bullet} = type_{R_q}(E_{|R_q|} - E_{|L_q|}) \\ \\ \text{Furthermore, for any edge } e \text{ in } T \text{ we define } ^{\bullet}e = \{q \in P : e \in q^{\bullet}\}, \underline{e} = \{q \in P : e \in q^{\bullet}\}, \underline{e} = \{q \in P : e \in q^{\bullet}\}, e^{\bullet} = \{q \in P : e \in q^{\bullet}\},$

An example of safe grammar can be found in Fig. 3 (for the moment ignore its relation to grammar CP in Fig. 2). For this grammar, **engage1** = $\{2:P, 3:P\}$, <u>engage1</u> = $\{1:CM\}$ and **engage1** = $\{5:PE, 6:PE\}$, while **engage1** = \emptyset , <u>1:CM</u> = \emptyset , <u>1:CM</u> = $\{engage1, engage2, engage3\}$ and $3:P^e = \{engage1, engage3\}$.

Definition 6 (Causal Relation). The causal relation of a safe grammar \mathcal{G} is the least transitive relation < over $Elem(\mathcal{G})$ satisfying, for any edge e in the graph of types T, and for productions $q, q' \in P$:

 $1. \ e \in {}^{\bullet}q \ \Rightarrow \ e < q; \qquad 2. \ e \in q^{\bullet} \ \Rightarrow \ q < e; \qquad 3. \ q^{\bullet} \cap q' \neq \emptyset \ \Rightarrow \ q < q'.$

As usual \leq is the reflexive closure of \leq . Moreover, for $x \in Elem(\mathcal{G})$ we denote by $\lfloor x \rfloor$ the set of causes of x in P, namely $\lfloor x \rfloor = \{q \in P : q \leq x\}$.

Note that the fact that an item is preserved by q and consumed by q', i.e., $\underline{q} \cap {}^{\bullet}q' \neq \emptyset$ does not imply q < q'. In this case, the dependency between the two productions is a kind of *asymmetric conflict* (see [4, 18, 16, 23]): The application of q' prevents q from being applied, so that q can neverfollow q' in a derivation (or, equivalently, if both q and q' occur in a derivation then q must precede q').

Definition 7 (Asymmetric Conflict). The asymmetric conflict \nearrow of a safe grammar \mathcal{G} is the relation over the set of productions P, defined by $q \nearrow q'$ if:

1. $q \cap {}^{\bullet}q' \neq \emptyset$; 2. ${}^{\bullet}q \cap {}^{\bullet}q' \neq \emptyset$ and $q \neq q'$; 3. q < q'.

Condition 1 is justified by the discussion above. Condition 2 essentially expresses the fact that the ordinary symmetric conflict is encoded, in this setting, as an asymmetric conflict in both directions. More generally, we will write q#q' and say that q and q' are in *conflict* when the causes of q and q', i.e., $\lfloor q \rfloor \cup \lfloor q' \rfloor$, includes a cycle of asymmetric conflict. Finally, since < represents a global order of execution, while \nearrow determines an order of execution only locally to each computation, it is natural to impose \nearrow to be an extension of < (Condition 3).

Definition 8 ((Nondeterministic) Occurrence Grammar). A (nondeterministic) occurrence grammar is a safe grammar $\mathcal{O} = \langle T, G_s, P, \pi \rangle$ such that

- 1. \leq is a partial order; for any $q \in P$, $\lfloor q \rfloor$ is finite and \nearrow is acyclic on $\lfloor q \rfloor$;
- 2. G_s is the graph graph($Min(\mathcal{O})$) generated by the set $Min(\mathcal{O})$ of minimal elements of $(Elem(\mathcal{O}), \leq)$, typed over T by the inclusion;
- 3. any item x in T is created by at most one production in P, i.e., $| \cdot x | \le 1$;
- 4. for each $q \in P$, the typing $type_{L_q}$ is injective on the "consumed" items in $|L_q| |R_q|$, and $type_{R_q}$ is injective on the "produced" items in $|R_q| |L_q|$.

Since the start graph of an occurrence grammar \mathcal{O} is determined by $Min(\mathcal{O})$, we often do not mention it explicitly.

Intuitively, Conditions 1–3 recast in the framework of graph grammars the conditions of occurrence nets (actually of occurrence contextual nets [4, 23]). In particular, in Condition 1, the acyclicity of asymmetric conflict on $\lfloor q \rfloor$ corresponds to the requirement of irreflexivity for the conflict relation in occurrence nets. Condition 4, instead, is closely related to safety and requires that each production consumes and produces items with multiplicity one. An example of an occurrence grammar is given in Fig. 3.

2.3 Concurrent Subgraphs, Configurations and Histories

The finite computations of an occurrence grammar are characterised by special subsets of productions closed under causal dependencies and with no conflicts (i.e., cycles of asymmetric conflict), suitably ordered.

Definition 9 (Configuration). Let $\mathcal{O} = \langle T, P, \pi \rangle$ be an occurrence grammar. A configuration of \mathcal{O} is a finite subset of productions $C \subseteq P$ such that \nearrow_C (the asymmetric conflict restricted to C) is acyclic, and for any $q \in C$, $\lfloor q \rfloor \subseteq C$. Given two configurations C_1 , C_2 we write $C_1 \sqsubseteq C_2$ if $C_1 \subseteq C_2$ and for any $q_1 \in C_1$, $q_2 \in C_2$, if $q_2 \nearrow q_1$ then $q_2 \in C_1$.

The set of all configurations of \mathcal{O} , ordered by \sqsubseteq , is denoted by $Conf(\mathcal{O})$.

Proposition 1 (Reachability of Graphs Generated by Configurations). Let \mathcal{O} be an occurrence grammar, $C \in Conf(\mathcal{O})$ be a configuration and

 $\mathbf{G}(C) = graph((Min(\mathcal{O}) \cup \bigcup_{q \in C} q^{\bullet}) - \bigcup_{q \in C} {}^{\bullet}q).$

Then a graph G such that $G \cong \mathbf{G}(C)$ can be obtained from the start graph of \mathcal{O} , by applying all the productions in C in any order compatible with \nearrow .

Due to the presence of asymmetric conflicts, given a production q, the history of q, i.e., the set of events which must precede q in a computation is not uniquely determined by q, but it depends also on the particular computation: the history of q can or can not include the productions in asymmetric conflict with q.

Definition 10 (History). Let \mathcal{O} be an occurrence grammar, let $C \in Conf(\mathcal{O})$ be a configuration and let $q \in C$. The history of q in C is the set of events $C[\![q]\!] = \{q' \in C : q' \nearrow_C^* q\}$. We denote by Hist(q) the set of histories of q, i.e., $Hist(q) = \{C[\![q]\!] : C \in Conf(\mathcal{O})\}.$

Reachable states can be characterised in terms of a concurrency relation.

Definition 11 (Concurrent Graph). Let $\mathcal{O} = \langle T, P, \pi \rangle$ be an occurrence grammar. A finite subset of edges $E \subseteq E_T$ is called concurrent, written co(E), if

1. \nearrow_E , the asymmetric conflict restricted to $\bigcup_{x \in E} [x]$, is acyclic;

2. $\neg (x < y)$ for all $x, y \in E$.

A subgraph G of T is called concurrent, written co(G), if $co(E_G)$.

It can be shown that the maximal concurrent subgraphs G of T correspond exactly (up to isolated nodes) to the graphs reachable from the start graph.

2.4 Unfolding of Graph Grammars

The unfolding construction, when applied to a grammar \mathcal{G} , produces a nondeterministic occurrence grammar $\mathcal{U}(\mathcal{G})$ describing the behaviour of \mathcal{G} . A construction for the double-pushout algebraic approach to graph rewriting has been proposed

in [3]: the one sketched here is simpler because productions cannot delete nodes and thus the dangling edge condition does not play a role.

The construction begins from the start graph of \mathcal{G} , and then applies in all possible ways its productions to concurrent subgraphs, recording in the unfolding each occurrence of production and each new graph item generated in the rewriting process.

Definition 12 (Unfolding - Sketch). Let $\mathcal{G} = \langle T, G_s, P, \pi \rangle$ be a graph grammar. The unfolding $\mathcal{U}(\mathcal{G}) = \langle T', G'_s, P', \pi' \rangle$ is the "componentwise" union of the following inductively defined sequence of occurrence grammars $\mathcal{U}(\mathcal{G})^{[n]}$.

 $(\mathbf{n} = \mathbf{0}) \ \mathcal{U}(\mathcal{G})^{[\mathbf{0}]}$ consists of the start graph $|G_s|$, with no productions.

 $(\mathbf{n} \to \mathbf{n} + \mathbf{1})$ Take $q \in P$ and let m be a match of q in the graph of types of $\mathcal{U}(\mathcal{G})^{[n]}$, satisfying the identification condition, such that $m(|L_q|)$ is concurrent.

Then the occurrence grammar $\mathcal{U}(\mathcal{G})^{[n+1]}$ is obtained by "recording" in $\mathcal{U}(\mathcal{G})^{[n]}$ the application of q at the match m. More precisely, a new production $q' = \langle q, m \rangle$ is added and the graph of types $T^{[n]}$ is extended by adding to it a copy of each item generated by the application q, without deleting any item.

The unfolding is mapped over the original grammar by the so-called folding morphism $\chi = \langle \chi_T, \chi_P \rangle : \mathcal{U}(\mathcal{G}) \to \mathcal{G}$. The first component $\chi_T : T' \to T$ is a graph morphism mapping each graph item in the (graph of types of) the unfolding to the corresponding item in the (graph of types of) the original grammar \mathcal{G} . The second component $\chi_P : P' \to P$ maps any production occurrence $\langle q, m \rangle$ in the unfolding to the corresponding production q of \mathcal{G} .

The occurrence grammar in Fig. 3 is an initial part of the (infinite) unfolding of the grammar CP in Fig. 2. For instance, production engage1 is an occurrence of production engage in CP, applied at the match consisting of the edges 1:CM, 2:P, 3:P. Unfolding such a match, three new graph items, two edges 5:PE, 6:PEand a node, are added to the graph of types of the unfolding. Note that the graph of types of the (partial) unfolding (call it T_T) is typed over the graph of types T_{CP} of the original grammar (via the folding morphism $\chi_T : T_T \to T_{CP}$). This explains why the edges of the graphs in the productions of the unfolding, which are typed over T_T , are marked with names including two colons.

The unfolding provides a compact representation of the behaviour of \mathcal{G} , and in particular it represents all the graphs reachable in \mathcal{G} , in the following sense.

Theorem 1 (Completeness of the Unfolding). Let $\mathcal{G} = \langle T, G_s, P, \pi \rangle$ be a graph grammar. A T-typed graph G is reachable in \mathcal{G} iff there exists a maximal concurrent subgraph X' of the graph of types of $\mathcal{U}(\mathcal{G})$ such that $G \simeq \langle X', \chi_{T|X'} \rangle$.

3 Finite Prefix for Graph Grammars

Let $\mathcal{G} = \langle T, G_s, P, \pi \rangle$ denote a graph grammar, fixed throughout the section, and let $\mathcal{U}(\mathcal{G}) = \langle T', P', \pi' \rangle$ be its unfolding with $\chi : \mathcal{U}(\mathcal{G}) \to \mathcal{G}$ the folding morphism, as in Definition 12. Given a configuration C of $\mathcal{U}(\mathcal{G})$, recall from Proposition 1 that $\mathbf{G}(C)$ denotes the subgraph of T' reached after the execution of the productions in C (up to isolated nodes). We shall denote by Reach(C) the same graph, seen as a graph typed over T by the restriction of the folding morphism, i.e., $Reach(C) = \langle \mathbf{G}(C), \chi_{T|\mathbf{G}(C)} \rangle$.

To identify a finite prefix of the unfolding the idea consists of avoiding to keep in the unfolding useless productions, i.e., productions which do not contribute to generating new graphs. The definition of "cut-off event" introduced by McMillan for Petri nets in order to formalise such a notion has to be adapted to this context, since for graph grammars a production may have different histories.

Definition 13 (Cut-Off). A production $q \in P'$ of the unfolding $\mathcal{U}(\mathcal{G})$ is a cutoff if there exists $q' \in P'$ such that $\operatorname{Reach}(\lfloor q \rfloor) \simeq \operatorname{Reach}(\lfloor q' \rfloor)$ and $|\lfloor q' \rfloor| < |\lfloor q \rfloor|$. A production q is a strong cut-off if for all $C_q \in \operatorname{Hist}(q)$ there exist $q' \in P'$ and $C_{q'} \in \operatorname{Hist}(q')$ such that $\operatorname{Reach}(C_q) \simeq \operatorname{Reach}(C_{q'})$ and $|C_{q'}| < |C_q|$. The truncation of \mathcal{G} is the greatest prefix $\mathcal{T}(\mathcal{G})$ of $\mathcal{U}(\mathcal{G})$ not including strong cut-offs.

Theorem 2 (Completeness and Finiteness of the Truncation). The truncation $\mathcal{T}(\mathcal{G})$ is a complete prefix of the unfolding, i.e., for any reachable graph G of \mathcal{G} there is a configuration C in $Conf(\mathcal{T}(\mathcal{G}))$ such that $Reach(C) \cong G$. Furthermore, if \mathcal{G} is *n*-bounded then the truncation $\mathcal{T}(\mathcal{G})$ is finite.

Unfortunately, the proof of the above theorem does not suggest a way of constructing the truncation for finite-state graph grammars. The problem essentially resides in the fact that the notion of strong cut-off refers to the set of histories of a production, which is, in general, infinite. While leaving the solution for the general case as an open problem, we next discuss how a finite complete prefix can be derived for a class of grammars for which this problem disappears. This still interesting class of graph grammars is characterised by a property that we call "read-persistence", since it appears as the graph grammar theoretical version of read-persistence as defined for contextual nets [23].

Definition 14 (**Read-Persistence**). An occurrence grammar $\mathcal{O} = \langle T, P, \pi \rangle$ is called read-persistent if for any $q_1, q_2 \in P$, if $q_1 \nearrow q_2$ then $q_1 \leq q_2$ or $q_1 \# q_2$. A graph grammar \mathcal{G} is called read-persistent if its unfolding $\mathcal{U}(\mathcal{G})$ is read-persistent.

It can be shown that an adaptation of the algorithm originally proposed in [17] for ordinary nets and extended in [23] to read-persistent contextual nets, works for read-persistent graph grammars. In particular, the notion of strong cut-off can be safely replaced by the weaker notion of (ordinary) cut-off. An obvious class of read-persistent graph grammars consists of all the grammars \mathcal{G} where any edge preserved by productions is never consumed.

For instance, the grammar CP in our running example is read-persistent, since the communication manager CM, the only edge preserved by productions, is never consumed. Its truncation is the graph grammar T(CP) depicted in Fig. 3. Denote by T_T its type graph. Note that applying the production [release] to any subgraph of T_T matching its left-hand side would result in a cut-off: this is the

reason why $\mathcal{T}(\mathcal{CP})$ does not include any instance of production [release]. The start graph of the truncation is isomorphic to the start graph of grammar \mathcal{CP} and it is mapped injectively to the graph of types $T_{\mathcal{T}}$ in the obvious way.



Fig. 3. The truncation $\mathcal{T}(\mathcal{CP})$ of the graph grammar in Fig. 2

In general, the truncation of a grammar such as CP where *n* processes are connected to *CM* in the start graph, will contain $\frac{n(n-1)}{2}$ productions. Considering instead all possible interleavings, we would end up with an exponential number of productions.

4 Exploiting the Prefix

In this section we propose a monadic second-order logic $\mathcal{L}2$ where some graph properties of interest can be expressed. Then we show how the validity of a property in $\mathcal{L}2$ over all the reachable graphs of a finite-state grammar \mathcal{G} can be verified by exploiting a complete finite prefix.

4.1 A Logic on Graphs

We first introduce the monadic second order logic $\mathcal{L}2$ for specifying graph properties. Quantification is allowed over edges, but not over nodes (as, e.g., in [8]).

Definition 15 (Graph Formulae). Let $\mathcal{X}_1 = \{x, y, ...\}$ be a set of (firstorder) edge variables and let $\mathcal{X}_2 = \{X, Y, ...\}$ be a set of (second-order) variables representing edge sets. The set of graph formulae of the logic $\mathcal{L}2$ is defined as follows, where $\ell \in \Lambda$, $i, j \in \mathbb{N}$:

$$F ::= x = y \mid c_i(x) = c_j(y) \mid type(x) = \ell \mid x \in X$$
(Predicates)
$$F \lor F \mid \neg F \mid \exists x.F \mid \exists X.F$$
(Connectives / Quantifiers)

TEAM LING


Fig. 4. The Petri net underlying the truncation $\mathcal{T}(\mathcal{CP})$ in Fig. 3

We denote by free(F) and Free(F) the sets of first-order and second-order variables, respectively, occurring free in F, defined in the obvious way.

Given a T-typed graph G, a formula F in $\mathcal{L}2$, and two valuations $\sigma : free(F) \rightarrow \mathcal{E}_{|G|}$ and $\Sigma : Free(F) \rightarrow \mathcal{P}(E_{|G|})$ for the free first- and second-order variables of F, respectively, the *satisfaction relation* $G \models_{\sigma,\Sigma} F$ is defined inductively, in the usual way; for instance $G \models_{\sigma,\Sigma} x = y$ iff $\sigma(x) = \sigma(y)$ and $G \models_{\sigma,\Sigma} x \in X$ iff $\sigma(x) \in \Sigma(X)$.

A simple, but fundamental observation is that, while for *n*-bounded graph grammars the graphical nature of the state plays a basic role, for any occurrence grammar \mathcal{O} we can can forget about it and view \mathcal{O} as an occurrence contextual net (i.e., a Petri net with read arcs, see, e.g., [4, 23]).

Definition 16 (Petri Net Underlying a Graph Grammar). The contextual Petri net underlying an occurrence grammar $\mathcal{O} = \langle T', P', \pi' \rangle$, denoted by Net(\mathcal{O}), is the Petri net having the set of edges $E_{T'}$ as places and a transition for every production $q \in P'$, with pre-set ${}^{\bullet}q$, post-set q^{\bullet} and context q.

For instance, the Petri net $Net(\mathcal{T}(\mathcal{CP}))$ underlying the truncation of \mathcal{CP} (see Fig. 3) is depicted in Fig. 4. Read arcs are represented as dotted undirected lines.

Let $\mathcal{G} = \langle T, G_s, P, \pi \rangle$ be a fixed finite-state graph grammar and consider the truncation $\mathcal{T}(\mathcal{G}) = \langle T', P', \pi' \rangle$ (actually, all the results hold for any complete finite prefix of the unfolding). Notice that, by completeness of $\mathcal{T}(\mathcal{G})$, any graph reachable in \mathcal{G} is (up to isolated nodes) a subgraph of the graph of types T' of $\mathcal{T}(\mathcal{G})$, typed over T by the restriction of the folding morphism $\chi : \mathcal{U}(\mathcal{G}) \to \mathcal{G}$. Also observe that a safe marking m of $Net(\mathcal{T}(\mathcal{G}))$ can be seen as a graph typed over the type graph T of the original grammar \mathcal{G} : take the least subgraph of T' having m as set of edges, i.e., graph(m), and type it over T by the restriction of the folding morphism. With a slight abuse of notation this typed graph will be denoted simply as graph(m).

We show how any formula ϕ in $\mathcal{L}2$ can be translated to a formula $M(\phi)$ over the safe markings of $Net(\mathcal{T}(\mathcal{G}))$ such that, for any marking *m* reachable in $Net(\mathcal{T}(\mathcal{G}))$

$$graph(m) \models \phi$$
 iff $m \models M(\phi)$.

95

The syntax of the formulae over markings is

$$\phi ::= e \mid \neg \phi \mid \phi \land \phi \mid \phi \lor \phi \mid \phi \to \phi,$$

where the basic formulae *e* are place (edge) names, meaning that the place is marked, i.e., $m \models e$ if $e \in m$. Logical connectives are treated as usual.

Definition 17 (Encoding Graph into Multiset Formulae). Let $\mathcal{T}(\mathcal{G})$ be the truncation of a graph grammar \mathcal{G} , as above. Let F be graph formula in $\mathcal{L}2$, let σ : free $(F) \rightarrow E_{T'}$ and Σ : Free $(F) \rightarrow \mathcal{P}(E_{T'})$. The encoding M is defined as:

$$\begin{split} M[x = y, \sigma, \Sigma] &= \text{true } if \ \sigma(x) = \sigma(y) \ and \ \text{false } otherwise \\ M[c_i(x) = c_j(y), \sigma, \Sigma] &= \begin{cases} \text{true } if \ |c_{T'}(\sigma(x))| \ge i \land |c_{T'}(\sigma(y))| \ge j \\ \land \ [c_{T'}(\sigma(x))]_i = [c_{T'}(\sigma(y))]_j \end{cases} \\ \text{false } otherwise \\ M[type(x) = \ell, \sigma, \Sigma] &= \text{true } if \ \chi_T(\sigma(x)) = \ell \ and \ \text{false } otherwise \\ M[x \in X, \sigma, \Sigma] &= \text{true } if \ \sigma(x) \in \Sigma(X) \ and \ \text{false } otherwise \\ M[F_1 \lor F_2, \sigma, \Sigma] &= M[F_1, \sigma, \Sigma] \lor M[F_2, \sigma, \Sigma] \\ M[\neg F, \sigma, \Sigma] &= \neg M[F, \sigma, \Sigma] \\ M[\exists x.F, \sigma, \Sigma] &= \bigvee_{e \in E_{T'}} (e \land M[F, \sigma \cup \{x \mapsto e\}, \Sigma]) \\ M[\exists X.F, \sigma, \Sigma] &= \bigvee_{E \subseteq E_{T'}, co(E)} (\bigwedge E \land M[F, \sigma, \Sigma \cup \{X \mapsto E\}]) \end{split}$$

where, for $E = \{e_1, \ldots, e_n\}$, the symbol $\bigwedge E$ stands for $e_1 \land \ldots \land e_n$. If F is closed formula (i.e., without free variables), we define $M[F] = M[F, \emptyset, \emptyset]$.

Note that, since every reachable graph in \mathcal{G} is isomorphic to a subgraph of T', typed by the restriction of χ_T , the encoding resolves the basic predicates by exploiting the structural information of T'. When a first-order variable x in a formula is mapped to an edge e, we take care that the edge is marked, and, similarly, when a second-order variable X in a formula is mapped to a set of edges E, such a set must be covered. Observe that in this case E is limited to range only over concurrent subsets of edges. In fact, if E is a non-concurrent set, then no reachable marking m will include E, i.e., $m \not\models \Lambda E$.

It is possible to show that the above encoding is correct, i.e., for any formula $\phi \in \mathcal{L}2$, for any pair of valuations $\sigma : \mathcal{X}_1 \to E_{T'}$ and $\Sigma : \mathcal{X}_2 \to \mathcal{P}(E_{T'})$, and for any safe marking *m* over $E_{T'}$, we have $graph(m) \models_{\sigma,\Sigma} \phi$ iff $m \models M[\phi, \sigma, \Sigma]$.

4.2 Checking Properties of Reachable Graphs

Let $\mathcal{G} = \langle G_s, T, P, \pi \rangle$ be a finite-state graph grammar. We next show how a complete finite prefix of \mathcal{G} can be used to check whether, given a formula $F \in \mathcal{L}2$, there exists some reachable graph which satisfies F. In this case we will write $\mathcal{G} \models \Diamond F$. The same algorithm allows to check "invariants" of a graph grammars, i.e., to verify whether a property $F \in \mathcal{L}2$ is satisfied by all graphs reachable in \mathcal{G} , written $\mathcal{G} \models \Box F$. In fact, it trivially holds that $\mathcal{G} \models \Box F$ iff $\mathcal{G} \not\models \Diamond \neg F$.

Let $\mathcal{T}(\mathcal{G}) = \langle T', P', \pi' \rangle$ be the truncation of \mathcal{G} (or any complete prefix of the unfolding) and let $Net(\mathcal{T}(\mathcal{G}))$ be the underlying Petri net. The formula produced by the encoding in Definition 17 can be simplified by exploiting the mutual relationships between items as expressed by the causality, (asymmetric) conflict and concurrency relation.

Proposition 2 (Simplification). Let *F* be any formula in $\mathcal{L}2$, let σ : free $(F) \rightarrow E_{T'}$ and Σ : Free $(F) \rightarrow \mathcal{P}(E_{T'})$ be valuations. If *m* is a marking reachable in Net $(\mathcal{T}(\mathcal{G}))$ and η is a marking formula obtained by simplifying $M[F, \sigma, \Sigma]$ with the Simplification Rule below:

If $S \subseteq E_{T'}$ and $\neg co(S)$ then replace the subformula $\bigwedge S$ by false.

then graph $(m) \models_{\sigma, \Sigma} F$ iff $m \models \eta$.

Algorithm. The question " $\mathcal{G} \models \Diamond F$?" is answered by working over $\mathsf{Net}(\mathcal{T}(\mathcal{G}))$:

- Consider the formula over markings M[F] (see Definition 17);
- Express M[F] in disjunctive normal form as below, where $a_{i,j}$ can be e or $\neg e$ for $e \in E_{T'}$:

$$\eta = \bigvee_{i=1}^n \bigwedge_{j=1}^{k_i} a_{i,j}$$

- Apply the Simplification Rule in Proposition 2, as far as possible, thus obtaining a formula η' ;
- For any conjunct in η' of the kind $e_1 \wedge \ldots \wedge e_h \wedge \neg e'_1 \wedge \ldots \wedge \neg e'_i$:
 - Take the configuration $C = \lfloor \{e_1, \ldots, e_h\} \rfloor$.
 - Consider the safe marking reached after C, i.e., $m_C = (m_0 \cup \bigcup_{t \in C} t^{\bullet}) \bigcup_{t \in C} t^{\bullet}$, where m_0 is the initial marking of $Net(T(\mathcal{G}))$ (consisting of all minimal places). Surely m_C includes $\{e_1, \ldots, e_h\}$. Hence, the only reason why the conjunct may not be true is that m_C includes some of the $\{e'_1, \ldots, e'_l\}$. In this case look for a configuration $C' \supseteq C$, which enriches C with transitions which consume the e'_j but not the e_i .
- The formula $\Diamond F$ holds iff this check succeeds for at least one conjunct.

For instance, suppose that we want to check that our sample graph grammar CP satisfies $\Box F$, where F is a L2 formula specifying that every engaged process is connected through connection c_2 to exactly one other engaged process, i.e.,

$$F = \forall x.(type(x) = PE \Rightarrow \exists y.(x \neq y \land type(y) = PE \land c_2(x) = c_2(y) \land \forall z.(type(z) = PE \land x \neq z \land c_2(x) = c_2(z) \Rightarrow y = z))).$$

The encoding $\phi = M[F]$ simplifies to

$$\phi \equiv (5: PE \iff 6: PE) \land (7: PE \iff 8: PE) \land (9: PE \iff 10: PE)$$

and we have to check that the truncation does not satisfy

$$\Diamond \neg \phi = \Diamond [(5: PE \land \neg 6: PE) \lor (\neg 5: PE \land 6: PE) \lor (7: PE \land \neg 8: PE) \\ \lor (\neg 7: PE \land 8: PE) \lor (9: PE \land \neg 10: PE) \lor (\neg 9: PE \land 10: PE)],$$

which can be done by using the described verification procedure.

97

5 Conclusions

We have discussed how the finite prefix approach, originally introduced by McMillan for Petri nets, can be generalised to graph transformation systems. A complete finite prefix can be constructed for some classes of graph grammars, but the problem of constructing it for general, possibly non-read-persistent grammars remains open and represents an interesting direction of further research. Also, it would be interesting to try to determine an upper bound on the size of the prefix, with respect to the number of reachable graphs.

We have shown how the complete finite prefix can be used to model-check some properties of interest for graph transformation systems. We plan to generalise the verification technique proposed here to allow the model-checking of more expressive logics, like the one studied in [10] for Petri nets, where temporal modalities can be arbitrarily nested. We intend to implement the model-checking procedure described in the paper and, as in the case of Petri nets, we expect that its efficiency could be improved by refined cut-off conditions (see, e.g., [11]) which help to decrease the size of the prefix.

As mentioned in the introduction, some efforts have been devoted recently to the development of suitable verification techniques for GTSs. A general theory of verification is presented in [12, 13], but without providing directly applicable techniques. In [15, 1, 5] one can find techniques which are applicable to infinitestate systems: the first defines a general framework based on types for graph rewriting, while the second is based on the construction of suitable approximations of the behaviour of a GTS. Instead, the papers [21, 19] concentrate on finite-state GTSs. They both generate a suitable labelled transition system out of a given finite-state GTS and then [21] resorts to model-checkers like SPIN, while [19] discusses the decidability of the model-checking problem for a logic, based on regular path expressions, allowing to talk about the history of nodes along computations. The main difference with respect to our work is that they do not exploit a partial order semantics, with an explicit representation of concurrency, and thus considering the possible interleavings of concurrent events these techniques may suffer of the state-explosion problem.

Acknowledgements. We would like to thank the anonymous referees for their helpful comments. We are also grateful to Javier Esparza for interesting and helpful discussions on the topic of this paper.

References

- P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. In *Proc. of CONCUR 2001*, pages 381–395. Springer, 2001. LNCS 2154.
- 2. P. Baldan, A. Corradini, and B. König. An unfolding-based approach for the verification of finite-state graph grammars. Technical report, Dipartimento di Informatica, Università Ca' Foscari di Venezia, 2004. To appear.

- P. Baldan, A. Corradini, and U. Montanari. Unfolding and Event Structure Semantics for Graph Grammars. In *Proc. of FoSSaCS '99*, pages 73–89. Springer, 1999. LNCS 1578.
- 4. P. Baldan, A. Corradini, and U. Montanari. Contextual Petri nets, asymmetric event structures and processes. *Information and Computation*, 171(1):1–49, 2001.
- P. Baldan, B. König, and B. König. A logic for analyzing abstractions of graph transformation systems. In *Proc. of SAS'03*, pages 255–272. Springer, 2003. LNCS 2694.
- A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Infor*maticae, 26:241–265, 1996.
- A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing* by Graph Transformation. Vol. 1: Foundations. World Scientific, 1997.
- 8. B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations.* World Scientific, 1997.
- 9. H. Ehrig, J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3: Concurrency, Parallelism and Distribution.* World Scientific, 1999.
- J. Esparza. Model checking using net unfoldings. Science of Computer Programming, 23(2–3):151–195, 1994.
- 11. J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20(20):285–310, 2002.
- 12. F. Gadducci, R. Heckel, and M. Koch. A fully abstract model for graph-interpreted temporal logic. In *Proc. of TAGT'98*, pages 310–322. Springer, 2000. LNCS 1764.
- R. Heckel. Compositional verification of reactive systems specified by graph transformation. In *Proc. of FASE'98*, pages 138–153. Springer, 1998. LNCS 1382.
- 14. G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
- 15. B. König. A general framework for types in graph rewriting. In *Proc. of FST TCS 2000*, pages 373–384. Springer, 2000. LNCS 1974.
- 16. R. Langerak. *Transformation and Semantics for LOTOS*. PhD thesis, Department of Computer Science, University of Twente, 1992.
- 17. K.L. McMillan. Symbolic Model Checking. Kluwer, 1993.
- G.M. Pinna and A. Poigné. On the nature of events: another perspective in concurrency. *Theoretical Computer Science*, 138(2):425–454, 1995.
- A. Rensink. Towards model checking graph grammars. In Proc. of the 3rd Workshop on Automated Verification of Critical Systems, Technical Report DSSE-TR-2003–2, pages 150–160. University of Southampton, 2003.
- 20. L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars.* PhD thesis, Technische Universität Berlin, 1996.
- D. Varró. Towards symbolic analysis of visual modelling languages. In Proc. GT-VMT 2002: International Workshop on Graph Transformation and Visual Modelling Techniques, volume 72.3 of Electronic Notes in Computer Science, pages 57–70. Elsevier, 2002.
- 22. W. Vogler. Efficiency of asynchronous systems and read arcs in Petri nets. In *Proc.* of *ICALP'97*, pages 538–548. Springer, 1997. LNCS 1256.
- 23. W. Vogler, A. Semenov, and A. Yakovlev. Unfolding and finite prefix for nets with read arcs. In *Proc. of CONCUR'98*, pages 501–516. Springer, 1998. LNCS 1466.

The Pros and Cons of Netcharts

Nicolas Baudru and Rémi Morin

Laboratoire d'Informatique Fondamentale de Marseille Université de Provence, 39 rue F. Joliot-Curie, F-13453 Marseille cedex 13, France

Abstract. Netcharts have been introduced recently by Mukund et al. in [17]. This new appealing approach to the specification of collections of message sequence charts (MSCs) benefits from a graphical description, a formal semantics based on Petri nets, and an appropriate expressive power. As opposed to high-level MSCs, any regular MSC language is the language of some netchart. Motivated by two open problems raised in [17], we establish in this paper that the questions

- (i) whether a given high-level MSC describes some netchart language
- $(\ensuremath{\textsc{ii}})$ whether a given netchart is equivalent to some high-level MSC

(iii) whether a given netchart describes a regular MSC language are undecidable. These facts are closely related to our first positive result: We prove that netchart languages are *exactly* the MSC languages that are implementable by message passing automata up to refinement of message contents. Next we focus on FIFO netcharts: The latter are defined as the netcharts whose executions correspond to *all* firing sequences of their low-level Petri net. We show that the questions

(i) whether a netchart is a FIFO netchart

(ii) whether a FIFO netchart describes a regular MSC language

(iii) whether a regular netchart is equivalent to some high-level MSC are decidable.

Introduction

Message Sequence Charts (MSCs) are a popular model often used for the documentation of telecommunication protocols. They profit by a standardized visual and textual presentation (ITU-T recommendation Z.120 [11]) and are related to other formalisms such as sequence diagrams of UML. An MSC gives a graphical description of communications between processes. It usually abstracts away from the values of variables and the actual contents of messages. However, this formalism can be used at a very early stage of design to detect errors in the specification [10]. In this direction, several studies have already brought up methods and complexity results for the model-checking and implementation of MSCs viewed as a specification language [1–3, 5, 8, 14, 16, 18, 19].

Collections of MSCs are often specified by means of high-level MSCs (HM-SCs). The latter can be seen as directed graphs labelled by component MSCs. However such specifications may be unrealistic because this formalism allows for the description of sets of MSCs that correspond to no communicating system. Furthermore in most cases *it is undecidable whether a HMSC describes*

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 99-114, 2004.

an implementable language [1, 14, 8]. In [17], Mukund et al. introduced a new formalism for specifying collections of MSCs: *Netcharts* can be seen as HMSCs with some distributed control whereas HMSCs require implicitly some global control over processes in the system. Basically a netchart is a Petri net whose places are labelled by processes and whose transitions are labelled by MSCs. This new approach benefits from a graphical description, a formal semantics, and an appropriate expressive power: As opposed to HMSCs, *all netcharts describe implementable languages*. Our first result completes this relationship and shows that *netcharts describe precisely all implementable languages* (Th. 3.7). This key result allows us to answer negatively to some questions left open in [17].

First we present several comparisons between netcharts and HMSCs. We show that *it is undecidable whether a HMSC describes a netchart language* (Th. 4.7). Conversely, we show also that *it is undecidable whether a netchart language can be described by some HMSC* (Cor. 4.4). Yet as explained below, we can effectively check whether a *regular* netchart is equivalent to some HMSC. These two results follow from the observation that a netchart language corresponds to some HMSC if and only if it describes a finitely generated set of MSCs (Th. 4.3).

In the literature regular MSC languages have attracted a lot of interest. These languages appeared in [2, 18] as a framework where many model-checking problems become decidable. They were investigated later thoroughly and characterized in a logical way in [8, 9]. In particular [8, Th. 4.1] shows how to decide whether a regular set of MSCs is finitely generated. Noteworthy, similarly to high-level *compositional* MSCs [7], any regular MSC language is the language of some netchart [17]. Answering a second open question from [17], another negative consequence of our first result is that *regularity is undecidable for netchart languages* (Cor. 3.8). This is admittedly a major drawback of netcharts.

Motivated by some restrictions considered at some point in [17], we prove in Theorem 5.3 that *regularity is decidable for the subclass of FIFO netcharts*. The latter are defined as those netcharts whose FIFO behaviors correspond exactly to the firing sequences of the underlying low-level Petri net. Theorem 5.3 relies on a difficult and unrecognized result by Lambert [12, Th. 5.2] together with the remark that a netchart language is regular if and only if it requires bounded channel capacities. Additionally we show that *we can check effectively whether a netchart is FIFO* (Th. 5.2) by reduction to the reachability problem in Petri nets [15].

This paper investigates two semantics of netcharts. The FIFO semantics adopted in [17] appears as a restriction of a more general semantics that allows non-FIFO behaviors. In most cases, results extend from the FIFO semantics to the non-FIFO one. However *we exhibit a netchart that is not implementable under the non-FIFO semantics*. To simplify the presentation of our results the non-FIFO semantics is investigated separately in the last section.

1 Background

Message sequence charts (MSCs) are defined by several recommendations that indicate how one should represent them graphically [11]. Examples of MSCs are



Fig. 1. FIFO MSC

Fig. 2. Non-FIFO MSC Fig. 3. Degenerate behavior

given in Figures 1 and 2 in which time flows top-down. In this paper we regard MSCs as particular labelled partial orders (or pomsets) following a traditional trend of modeling concurrent executions [6,13,20]. This approach allows for applying classical results of Mazurkiewicz trace theory to the framework of MSCs [18,8,9,16,3].

A *pomset* over an alphabet Σ is a triple $t = (E, \preccurlyeq, \xi)$ where (E, \preccurlyeq) is a finite partial order and ξ is a mapping from E to Σ . A pomset can be seen as an abstraction of an execution of a concurrent system. In this view, the elements e of E are *events* and their label $\xi(e)$ describes the basic action of the system that is performed by the event $e \in E$. Furthermore, the order \preccurlyeq describes the causal dependence between the events.

An order extension of a pomset $t = (E, \preccurlyeq, \xi)$ is a pomset $t' = (E, \preccurlyeq', \xi)$ such that $\preccurlyeq \subseteq \preccurlyeq'$. A *linear extension* of t is an order extension that is linearly ordered. It corresponds to a sequential view of the concurrent execution t. Linear extensions of a pomset t over Σ can naturally be regarded as words over Σ . By $LE(t) \subseteq \Sigma^*$, we denote the set of linear extensions of a pomset t over Σ .

1.1 FIFO and Non-FIFO Basic Message Sequence Charts

We present here a formal definition of MSCs. The latter appear as particular pomsets over some alphabet $\Sigma_{\mathcal{I}}^{A}$ that we introduce first. Let \mathcal{I} be a finite set of processes (also called *instances*) and Λ be a finite set of messages. For any instance $i \in \mathcal{I}$, the alphabet $\Sigma_{i}^{\Lambda} = \Sigma_{1,i}^{\Lambda} \cup \Sigma_{7,i}^{\Lambda}$ is the disjoint union of the set of send actions $\Sigma_{1,i}^{\Lambda} = \{i!^{x}j \mid j \in \mathcal{I} \setminus \{i\}, x \in \Lambda\}$ and the set of *receive actions* $\Sigma_{7,i}^{\Lambda} = \{i?^{x}j \mid j \in \mathcal{I} \setminus \{i\}, x \in \Lambda\}$. The alphabets Σ_{i}^{Λ} are disjoint and we put $\Sigma_{\mathcal{I}}^{\Lambda} = \bigcup_{i \in \mathcal{I}} \Sigma_{i}^{\Lambda}$. Given an action $a \in \Sigma_{\mathcal{I}}^{\Lambda}$, we denote by $\operatorname{Ins}(a)$ the unique instance i such that $a \in \Sigma_{i}^{\Lambda}$, that is the particular instance on which each occurrence of action a takes place.

For any poinset (E, \preccurlyeq, ξ) over $\Sigma_{\mathcal{I}}^{A}$ we denote by $\operatorname{Ins}(e)$ the instance on which the event e occurs: $\operatorname{Ins}(e) = \operatorname{Ins}(\xi(e))$. We say that f covers e and we write $e \rightarrow f$ if $e \prec f$ and $e \prec g \preccurlyeq f$ implies g = f. We say that two events e and fare two matching events and we write $e \rightsquigarrow f$ if e is the *n*-th send event $i!^{x}j$ and f is the *n*-th receive event $j?^{x}i$: In other words, we put $e \rightsquigarrow f$ if there are two instances i and j and some message $x \in \Lambda$ such that $\xi(e) = i!^{x}j, \xi(f) = j?^{x}i$ and $\operatorname{Card}\{e' \in E \mid \xi(e') = i!^{x}j \land e' \preccurlyeq e\} = \operatorname{Card}\{f' \in E \mid \xi(f') = j?^{x}i \land f' \preccurlyeq f\}$. **Definition 1.1.** A basic message sequence chart (MSC) over the set of messages A is a pomset $M = (E, \preccurlyeq, \xi)$ over Σ_T^A such that $M_1: \forall e, f \in E: \operatorname{Ins}(e) = \operatorname{Ins}(f) \Rightarrow (e \preccurlyeq f \lor f \preccurlyeq e)$ $M_2: \forall e \in E, \exists f \in E, e \rightsquigarrow f \lor f \rightsquigarrow e$ $M_3: e \rightsquigarrow f \Rightarrow e \preccurlyeq f$ $M_4: [e \longrightarrow f \land \operatorname{Ins}(e) \neq \operatorname{Ins}(f)] \Rightarrow e \rightsquigarrow f.$

By M_1 , events occurring on the same instance are linearly ordered: In particular non-deterministic choice cannot be described within a basic MSC. Condition M_2 makes sure that each receive event matches some send event and conversely. Thus there is no duplication of messages within the channels and M_2 formalizes partly the reliability of the channels. Following the recommendation Z.120, we allow *overtaking* (Fig. 2) but forbid any reversal of the order in which two identical messages m sent from i to j are received by j (Fig. 3). Now M_3 formalizes simply that the receipt of any message will occur after the corresponding send event. Finally, by M_4 , causality in M consists only in the linear dependency on each instance and the ordering of pairs of matching events. The set of all basic MSCs is denoted by bMSC. Note here that if two basic MSCs share some linear extension then they are equal. We denote by Ins(M) the set of active instances of an MSC M: $Ins(M) = \{i \in \mathcal{I} \mid \exists e \in E, Ins(e) = i\}$.

In Figure 2, the basic MSC exhibits some *overtaking* of message y above two messages x. A basic MSC satisfies *the FIFO requirement* if it shows no overtaking, that is, the messages from one instance to another are delivered in the order they are sent (Fig. 1). Non-FIFO basic MSCs allow for specifying scenarios that use several channels (or message types) between pairs of processes (Fig. 2). A more critical situation is illustrated by Figure 3. In this drawing, one message overtakes another one with the same content: In this paper, differently from [4] we forbid this kind of behaviors.

1.2 Petri Nets

Let us now recall the definition of a Petri net and some usual notations. A *Petri* net is a triple $\mathcal{P} = (P, T, F)$ where P is a set of places, T is a set of transitions such that $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (T \times P)$ is a flow relation. We shall use the following usual notations. For all $x \in P \cup T$, we put $\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$ and $x^{\bullet} = \{y \in P \cup T \mid (x, y) \in F\}$. Clearly, for all transitions $t, \bullet t$ and t^{\bullet} are sets of places, and conversely for all places $p \in P, \bullet p$ and p^{\bullet} are both sets of transitions. A marking \mathfrak{m} of \mathcal{P} is a multiset of places $\mathfrak{m} \in P^{\mathbb{N}}$. A transition t is enabled at $\mathfrak{m} \in P^{\mathbb{N}}$ if $\mathfrak{m}(p) \ge 1$ for all $p \in \bullet t$. In this case, we write $\mathfrak{m}[t\rangle \mathfrak{m}'$ where the marking \mathfrak{m}' is defined by $\mathfrak{m}'(p) = \mathfrak{m}(p) - 1$ if $p \in \bullet t \setminus t^{\bullet}, \mathfrak{m}'(p) = \mathfrak{m}(p) + 1$ if $p \in t^{\bullet} \setminus \bullet t$, and $\mathfrak{m}'(p) = \mathfrak{m}(p)$ otherwise.

In this paper, we consider Petri nets provided with an *initial marking* \mathfrak{m}_{in} and a *finite* set of final markings \mathfrak{F} . An *execution sequence* is a word $u = t_1 \dots t_n \in T^*$ such that there are markings $\mathfrak{m}_0, \dots, \mathfrak{m}_n$ satisfying $\mathfrak{m}_0 = \mathfrak{m}_{in}, \mathfrak{m}_n \in \mathfrak{F}$ and $\mathfrak{m}_{k-1}[t_k) \mathfrak{m}_k$ for all integers $k \in [1, n]$. The language $L(\mathcal{P})$ consists of all execution sequences of \mathfrak{P} .



Fig. 4. A netchart $\ensuremath{\mathbb{N}}$ and a corresponding MSC

1.3 Netcharts

A netchart is basically a Petri net whose places are labelled by instances and whose transitions are labelled by FIFO basic MSCs. Similarly to Petri nets, netcharts admit an intuitive visual representation: Examples of netcharts are given in Fig. 4, 7, and 9.

Definition 1.2. A netchart over Λ consists of a Petri net $(P, T, F, \mathfrak{m}_{in}, \mathfrak{F})$ and two mappings $\text{Ins} : P \to \mathcal{I}$ and $\mathcal{M} : T \to \text{bMSC}$ such that Ins associates some instance Ins(p) to each place p and \mathcal{M} associates a FIFO basic $MSC \mathcal{M}(t)$ over the set of messages Λ to each transition $t \in T$. Three conditions are required for such a structure to be a netchart:

- N₁: For each instance $i \in \mathcal{I}$, the places located on instance i contain exactly one token in the initial marking, i.e. $\sum_{\text{Ins}(p)=i} \mathfrak{m}_{\text{in}}(p) = 1$.
- N₂: For each transition t and each active instance $i \in \text{Ins}(\mathcal{M}(t))$, there is exactly one place $p \in {}^{\bullet}t$ such that Ins(p) = i and there is exactly one place $p \in t^{\bullet}$ such that Ins(p) = i.
- N₃: For each transition t and each place $p \in {}^{\bullet}t \cup t^{\bullet}$, the instance associated to p is active in $\mathcal{M}(t)$: $\operatorname{Ins}(p) \in \operatorname{Ins}(\mathcal{M}(t))$.

Observe here that the last requirement N_3 implies that ${}^{\bullet}t \cup t^{\bullet}$ is empty as soon as $\mathcal{M}(t)$ is the empty MSC. However Axiom N_3 plays actually no rôle in the semantics of netcharts and it could be removed for simplification's sake.

2 Semantics of Netcharts

In this section we fix a netchart $\mathbb{N} = ((P, T, F, \mathfrak{m}_{in}, \mathfrak{F}), \operatorname{Ins}, \mathcal{M})$ over the set of messages Λ and define formally its behaviors. The semantics of \mathbb{N} consists of basic MSCs over Λ (Fig. 4). The latter are derived from the basic MSCs that represent the execution sequences of some low-level Petri net $\mathcal{P}_{\mathbb{N}}$ (Fig. 4 and 6). Actually, the execution sequences of $\mathcal{P}_{\mathbb{N}}$ use a *refined set of messages* Λ° and MSCs of \mathbb{N} are obtained by projection of messages from Λ° onto Λ .

2.1 From MSCs to Petri Nets

The construction of the low-level Petri net $\mathcal{P}_{\mathcal{N}}$ starts with the translation of each transition $t \in T$ with component MSC $\mathcal{M}(t) = (E, \preccurlyeq, \xi)$ into some Petri net $\mathcal{P}_t = (P_t, T_t, F_t)$. This natural operation is depicted in Fig. 5.

This construction need to regard each component MSC $M = (E, \preccurlyeq, \xi)$ as a dag (direct acyclic graph) denoted by (E, \prec, ξ) . For any instance $i \in \mathcal{I}$ we let \preccurlyeq_i be the restriction of \preccurlyeq to events located on instance *i*. Then $e \rightarrow _i f$ if *e* occurs immediately before *f* on instance *i*. Then the binary relation \prec consists of pairs of matching events together with pairs of covering events w.r.t. \preccurlyeq_i .

Definition 2.1. The MSC dag of a basic MSC $M = (E, \preccurlyeq, \xi)$ is a labelled directed acyclic graph (E, \preccurlyeq, ξ) such that $e \preccurlyeq f$ if $e \rightsquigarrow f$ or $e \prec_i f$ for some instance $i \in \mathcal{I}$.

Clearly we can recover the basic MSC from its MSC dag. The reason for this is that $\neg \subseteq \neg$ hence $\neg \equiv$ is simply the reflexive and transitive closure of \neg . That is why we will identify a basic MSC with its corresponding MSC dag.

We can now formalize how each component MSC $\mathcal{M}(t) = (E, \prec, \xi)$ is translated into some Petri net $\mathcal{P}_t = (P_t, T_t, F_t)$. First, the places P_t are identified with pairs from \prec . Second the transitions T_t are identified with some send or receive actions over a new set of messages from $A \times T \times P_t$. Formally, we put $P_t = \prec$ and $T_t = \{i!^{m,t,(e,f)}j, j?^{m,t,(e,f)}i \mid (e, f) \in \prec \land \xi(e) = i!^m j \land \xi(f) = j?^m i\}$.

Note that the translation from the basic MSC $\mathcal{M}(t)$ into the Petri net \mathcal{P}_t is one-to-one: We will be able to recover the basic MSC $\mathcal{M}(t)$ from the Petri net \mathcal{P}_t . For this, we let ρ be the mapping from T_t to E such that $\rho(i!^{m,t,(e,f)}j) = e$ and $\rho(j?^{m,t,(e,f)}i) = f$. To complete the definition of \mathcal{P}_t we choose a flow relation F_t in accordance with the causality relation \prec of $\mathcal{M}(t)$: We put

$$F_t = \{ (r, (e, f)) \in T_t \times P_t \mid \rho(r) = e \} \cup \{ ((e, f), r) \in P_t \times T_t \mid \rho(r) = f \}.$$

The transitions of the Petri net $\mathcal{P}_t = (P_t, T_t, F_t)$ will be connected to places of \mathcal{N} by means of the following connection relation:

$$F'_t = \{(p,r) \in P \times T_t \mid p \in {}^{\bullet}t \land {}^{\bullet}r = \emptyset \land \operatorname{Ins}(\rho(r)) = \operatorname{Ins}(p)\} \\ \cup \{(r,p) \in T_t \times P \mid p \in t^{\bullet} \land r^{\bullet} = \emptyset \land \operatorname{Ins}(\rho(r)) = \operatorname{Ins}(p)\}.$$

2.2 Low-Level Petri Net

Now, in order to build the low-level Petri net $\mathcal{P}_{\mathcal{N}}$ of the netchart \mathcal{N} , we replace each transition $t \in T$ of \mathcal{N} by its corresponding Petri net \mathcal{P}_t as shown in Fig. 6.



Fig. 5. From transition t_1 to Petri net \mathcal{P}_{t_1}



Fig. 6. The low-level Petri net $\mathcal{P}_{\mathcal{N}}$ associated to the netchart \mathcal{N} of Fig. 4

The low-level Petri net $\mathcal{P}_{\mathcal{N}} = (P_{\mathcal{N}}, T_{\mathcal{N}}, F_{\mathcal{N}}, \mathfrak{m}_{\text{in}}, \mathfrak{F}_{\mathcal{N}})$ is built as follows. First, the set of places $P_{\mathcal{N}}$ collects the places of \mathcal{N} and the places of all $\mathcal{P}_t: P_{\mathcal{N}} = \bigcup_{t \in T} P_t \cup P$. Second, the set of transitions collects all transitions of all $\mathcal{P}_t: T_{\mathcal{N}} = \bigcup_{t \in T} T_t$. Now the flow relation consists of the flow relation F_t of each \mathcal{P}_t together with the connection relations $F'_t: F_{\mathcal{N}} = \bigcup_{t \in T} F_t \cup F'_t$. The initial marking of \mathcal{P} is the one of \mathcal{N} : The new places $p \in P_{\mathcal{N}} \setminus P$ are initially empty. Similarly a marking \mathfrak{m} of $\mathcal{P}_{\mathcal{N}}$ is final if the restriction of \mathfrak{m} to the places of \mathcal{N} is a final marking of \mathcal{N} and if all other places are empty: $\mathfrak{F}_{\mathcal{N}} = \{\mathfrak{m} \in P^{\mathcal{N}} \mid \mathfrak{m}_{IP} \in \mathfrak{F} \land \mathfrak{m}_{IP_{\mathcal{N}}}\} = 0\}$.

and if all other places are empty: $\mathfrak{F}_{\mathcal{N}} = \{\mathfrak{m} \in P^{\mathbb{N}} \mid \mathfrak{m}_{|P} \in \mathfrak{F} \land \mathfrak{m}_{|P_{\mathcal{N}} \setminus P} = 0\}$. Any execution sequence $u \in L(\mathcal{P}_{\mathcal{N}})$ of the low-level Petri net leads from the initial marking to some final marking for which all places from P_t are empty. Moreover u is actually a linear extension of a unique basic MSC.

Definition 2.2. The MSC language $\mathcal{L}_{fifo}(\mathfrak{P}_{N})$ consists of the FIFO basic MSCs M such that at least one linear extension of M is an execution sequence of \mathfrak{P}_{N} .

Interestingly, similarly to a property observed with message passing automata (Def. 3.2 below), it can be easily shown that a basic MSC M belongs to $\mathcal{L}_{fifo}(\mathcal{P}_{N})$ if and only if *all* linear extensions of M are execution sequences of \mathcal{P}_{N} . Noteworthy it can happen that an execution sequence of the low-level Petri net \mathcal{P}_{N} corresponds to a *non-FIFO* MSC (see e.g. [17, Pig. 5] or Fig. 7). Following [17], we focus on FIFO behaviors and neglect this kind of execution sequences here. We will investigate a non-FIFO semantics of netcharts in the last section only.

2.3 Set of MSCs Associated to Some Netchart

Recall now that MSCs from $\mathcal{L}_{fifo}(\mathcal{P}_{N})$ use a refined set of messages Λ° that consists of triples (m, t, a) where $m \in \Lambda$, $t \in T$, and $a \in P_{t}$. We let $\pi^{\circ} : \Lambda^{\circ} \to \Lambda$ denote the labelling that associates the message $m \in \Lambda$ to each triple $(m, t, a) \in \Lambda^{\circ}$. This labelling extends to a function that maps actions of $\Sigma_{\mathcal{I}}^{\Lambda^{\circ}}$ onto actions of $\Sigma_{\mathcal{I}}^{\Lambda}$ in a natural way. Furthermore this mapping extends in the obvious way from the FIFO basic MSCs over Λ° to the FIFO basic MSCs over Λ . The semantics of the netchart \mathcal{N} is defined from the semantics of its low-level Petri net $\mathcal{P}_{\mathcal{N}}$ by means of the labelling π° .



Fig. 7. Netchart \mathcal{N}_1 and some non-FIFO behaviour $N \notin \mathcal{L}_{\text{fifo}}(\mathcal{N})$

Definition 2.3. The MSC language $\mathcal{L}_{fifo}(\mathcal{N})$ is the set of FIFO basic MSCs obtained from an MSC of its low-level Petri net by the labelling $\pi^{\circ}: \mathcal{L}_{fifo}(\mathcal{N}) = \pi^{\circ}(\mathcal{L}_{fifo}(\mathcal{P}_{\mathcal{N}})).$

We stress here that π° maps FIFO basic MSCs onto FIFO basic MSCs. The situation with non-FIFO basic MSCs may be more complicated as we will see in the last section.

3 Netcharts vs. Implementable Languages

In this section, we study how netcharts relate to communicating systems. We consider the set of channels \mathcal{K} that consists of all triples $(i, j, x) \in \mathcal{I} \times \mathcal{I} \times \Lambda$: A *channel state* is then formalized by a map $\chi : \mathcal{K} \to \mathbb{N}$ that describes the queues of messages within the channels at some stage of an execution. The *empty channel state* χ_0 is such that each channel maps to 0.

Definition 3.1. A message passing automaton (MPA) \$ over Λ consists of a family of local components $(\mathcal{A}_i)_{i \in \mathcal{I}}$ and a subset of global final states F such that each component \mathcal{A}_i is a transition system $(Q_i, \iota_i, \longrightarrow_i)$ over Σ_i^{Λ} where Q_i is a finite set of *i*-local states, with initial state $\iota_i \in Q_i, \longrightarrow_i \subseteq (Q_i \times \Sigma_i^{\Lambda} \times Q_i)$ is the *i*-local transition relation and $F \subseteq (\prod_{i \in \mathcal{I}} Q_i) \times {\chi_0}$.

3.1 Semantics of MPA

A global state is a pair (s, χ) where $s \in \prod_{i \in \mathcal{I}} Q_i$ is a tuple of local states and χ is a channel state. The *initial global state* is the pair $i = (s, \chi)$ such that $s = (\iota_i)_{i \in \mathcal{I}}$ and $\chi = \chi_0$ is the empty channel state. The system of global states associated to S is the transition system $\mathcal{A}_S = (Q, \iota, \longrightarrow)$ over $\Sigma_{\mathcal{I}}^{\Lambda}$ where $Q = \prod_{i \in \mathcal{I}} Q_i \times \mathbb{N}^{\mathcal{K}}$ is the set of global states and the global transition relation $\longrightarrow \subseteq Q \times \Sigma_{\mathcal{I}}^{\Lambda} \times Q$ satisfies:

- for all
$$(i, j, m) \in \mathcal{K}$$
, $(q_k)_{k \in \mathcal{I}}, \chi \xrightarrow{i!^m j} (q'_k)_{k \in \mathcal{I}}, \chi'$ if
1. $q_i \xrightarrow{i!^m j}_{i \neq i} q'_i$ and $q'_k = q_k$ for all $k \in \mathcal{I} \setminus \{i\}$,
2. $\chi'(i, j, m) = \chi(i, j, m) + 1$ and $\chi(x) = \chi'(x)$ for all $x \in \mathcal{K} \setminus \{(i, j, m)\}$;
- for all $(i, j, m) \in \mathcal{K}$, $(q_k)_{k \in \mathcal{I}}, \chi \xrightarrow{j?^m i} (q'_k)_{k \in \mathcal{I}}, \chi'$ if
1. $q_j \xrightarrow{j?^m i}_{j \neq j} q'_j$ and $q'_k = q_k$ for all $k \in \mathcal{I} \setminus \{j\}$,
2. $\chi(i, j, m) = 1 + \chi'(i, j, m)$ and $\chi(x) = \chi'(x)$ for all $x \in \mathcal{K} \setminus \{(i, j, m)\}$.

TEAM LING

As usual with transition systems, for any $u = a_1 \dots a_n \in \Sigma_{\mathcal{I}}^{\Lambda^*}$, we write $q \xrightarrow{u} q'$ if there are some global states $q_0, \dots, q_n \in Q$ such that $q_0 = q, q_n = q'$ and for all $r \in [1, n], q_{r-1} \xrightarrow{a_r} q_r$. An execution sequence of S is a word $u \in \Sigma_{\mathcal{I}}^{\Lambda^*}$ such that $i \xrightarrow{u} q$ for some global final state $q \in F$.

Consider now an MPA S with components $(\mathcal{A}_i)_{i \in \mathcal{I}}$ and global final states F. Any execution sequence $u \in \Sigma_{\mathcal{I}}^*$ is a linear extension of a (unique) basic MSC.

Definition 3.2. The language $\mathcal{L}_{fifo}(S)$ consists of the FIFO basic MSCs M such that at least one linear extension of M is an execution sequence of S.

Noteworthy, it can be easily shown that a basic MSC M belongs to $\mathcal{L}_{fifo}(S)$ iff *all* linear extensions of M are execution sequences of S. We say that a language $\mathcal{L} \subseteq bMSC$ is *realizable* if there exists some MPA S such that $\mathcal{L} = \mathcal{L}_{fifo}(S)$.

Example 3.3. Consider the netchart \mathcal{N}_1 depicted in Figure 7 for which the initial marking is the single final marking. Its language $\mathcal{L}_{\text{fifo}}(\mathcal{N}_1)$ is the set of all basic MSCs that consist only of messages *a* and *b* exchanged from *i* to *j* in a FIFO manner. Clearly, the language $\mathcal{L}_{\text{fifo}}(\mathcal{N}_1)$ is realizable.

3.2 Implementation of MSC Languages

As observed in [1], there are finite sets of FIFO basic MSCs that are not realizable. For this reason, it is natural to relax the notion of realization. In [9], Henriksen et al. suggested to allow some refinements of message contents as follows.

Definition 3.4. Let $\mathcal{L} \subseteq \mathbb{bMSC}$ be an MSC language over the set of messages A. We say that \mathcal{L} is implementable if there are some MPA \mathcal{S} over some set of messages Λ' and some labelling $\lambda : \Lambda' \to \Lambda$ such that $\mathcal{L} = \lambda(\mathcal{L}_{flo}(\mathcal{S}))$.

Note here that any implementable language consists of FIFO basic MSCs only because $\lambda(M)$ is FIFO as soon as M is FIFO.

As the next result shows, the refinement of message contents by means of labellings helps the synthesis of MPAs from sets of scenarios. As opposed to the restrictive approach studied in [1,14] which sticks to the specified set of message contents, labellings allow for the implementation of any finite set of basic MSCs. Actually the refinement of messages allows for the implementation of any *regular* set of FIFO basic MSCs. Recall here that an MSC language $\mathcal{L} \subseteq \text{bMSC}$ is called *regular* if the set of corresponding linear extensions $\text{LE}(\mathcal{L}) = \{\text{LE}(M) \mid M \in \mathcal{L}\}$ is a regular set of words.

Theorem 3.5. [9, Th. 3.4] All regular sets of FIFO basic MSCs are implementable.

One main property of netcharts is the following.

Theorem 3.6. [17] For any netchart \mathbb{N} , $\mathcal{L}_{fifo}(\mathbb{N})$ is implementable.

Note that Theorem 3.6 fails if we forbid refinements, that is if we require that $\mathcal{L}_{fifo}(\mathcal{N}) = \mathcal{L}_{fifo}(\mathfrak{S})$. The reason for this is again that there are finite sets of FIFO basic MSCs that are not realizable while they are netchart languages.

3.3 From Message Passing Automata to Netcharts

In [17, Th. 6], it is shown that any regular MSC language is a netchart language. However the converse fails: There are netchart languages that are not regular (see e.g. Example 3.3). Our first result characterizes the expressive power of netcharts and establishes the converse of Theorem 3.6.

Theorem 3.7. Any implementable language is the MSC language of some netchart whose component MSCs consist only of a pair of matching events.

We stress that Theorem 3.7 is effective: For any MPA S over the set of messages Λ' and any labelling $\lambda : \Lambda' \to \Lambda$, we can build a netchart \mathcal{N} such that $\mathcal{L}_{flfo}(\mathcal{N}) = \lambda(\mathcal{L}_{flfo}(S))$. Theorem 3.7 subsumes [17, Th. 6] because all regular MSC languages are implementable (Th. 3.5) and there are implementable languages that are not regular (Ex. 3.3). The proof of Theorem 3.7 is rather tedious. It differs from the proof of [17, Th. 6] in that we do not assume the implementable language $\lambda(\mathcal{L}_{flfo}(S))$ to be regular.

Theorem 3.7 shows that the expressivity of netcharts coincides with the expressivity of MPAs up to labellings. This leads us to a first answer to questions from [17].

Corollary 3.8. It is undecidable whether a netchart language is regular.

Proof. We observe first that it is undecidable whether the language of some given MPA is regular. More precisely, similarly to the proof of [19, Prop. 7], for any instance of Post's Corresponding Problem, we build some MPA \$ such that the instance has a solution iff $\mathcal{L}_{fifo}(\$)$ is not empty and in this case $\mathcal{L}_{fifo}(\$)$ is not regular. Now the proof follows from the effectiveness of Th. 3.7 with a labelling $\lambda = \text{Id} : \Lambda \to \Lambda$.

4 Netcharts vs. High-Level Message Sequence Charts

Let us now recall how one can build high-level MSCs from basic MSCs. First, the asynchronous concatenation of two basic MSCs $M_1 = (E_1, \preccurlyeq_1, \xi_1)$ and $M_2 = (E_2, \preccurlyeq_2, \xi_2)$ is the basic MSC $M_1 \cdot M_2 = (E, \preccurlyeq, \xi)$ where $E = E_1 \uplus E_2$, $\xi = \xi_1 \cup \xi_2$ and the partial order \preccurlyeq is the transitive closure of $\preccurlyeq_1 \cup \preccurlyeq_2 \cup \{(e_1, e_2) \in E_1 \times E_2 \mid \text{Ins}(e_1) = \text{Ins}(e_2)\}$. This concatenation allows for the composition of specifications in order to describe infinite sets of basic MSCs: We obtain high-level message sequence charts as rational expressions, following thus the usual algebraic approach that we recall next.

4.1 Rational Sets of MSCs

For any subsets \mathcal{L} and \mathcal{L}' of bMSC, the *product* of \mathcal{L} by \mathcal{L}' is $\mathcal{L} \cdot \mathcal{L}' = \{x \cdot x' \mid x \in \mathcal{L} \land x' \in \mathcal{L}'\}$. We let 1 denote the empty basic MSC and we put $\mathcal{L}^0 = \{1\}$. For any $n \in \mathbb{N}$, $\mathcal{L}^{n+1} = \mathcal{L}^n \cdot \mathcal{L}$; then the *iteration* of \mathcal{L} is $\mathcal{L}^* = \bigcup_{n \in \mathbb{N}} \mathcal{L}^n$. It is also denoted $\langle \mathcal{L} \rangle$. A language $\mathcal{L} \subseteq$ bMSC is *finitely generated* if there is a finite subset Γ of bMSC such that $\mathcal{L} \subseteq \langle \Gamma \rangle$. A subset of bMSC is *rational* if it can be obtained from the finite subsets of bMSC by means of unions, products and iterations. Any rational language is finitely generated.

Definition 4.1. A high-level message sequence chart (HMSC) is a rational expression of basic MSCs, that is, an expression built from finite sets of basic MSCs by use of union (+), product (\cdot) and iteration (\star) .

We follow here the approach adopted e.g. in [1,2,5,8,14,19] where HMSCs are however often flattened into *message sequence graphs*. The set of MSCs corresponding to some HMSC \mathcal{H} is denoted by $\mathcal{L}_{\mathcal{H}}$.

Example 4.2. Consider again the two components MSCs A and B of the netchart \mathcal{N}_1 depicted in Fig. 7. As already observed in Example 3.3, the language $\mathcal{L}_{\text{fifo}}(\mathcal{N}_1)$ is the set of all FIFO basic MSCs that consist only of messages a and b exchanged from i to j. This language corresponds to the HMSC $(A+B)^*$.

4.2 For Netchart Languages: Finitely Generated Means Rational

As already observed in [17, Fig. 6], there are netcharts whose languages are not finitely generated. Clearly these netchart languages are not rational. We show here that it is undecidable whether a given netchart language is described by some HMSC (Cor. 4.4). As a first step, the next result shows that being finitely generated is sufficient for a netchart language to be rational.

Theorem 4.3. For any netchart \mathbb{N} , $\mathcal{L}_{fifo}(\mathbb{N})$ is finitely generated iff it is the language of some HMSC.

Proof. Let Γ be a finite set of basic MSCs over Λ such that $\mathcal{L}_{fifo}(\mathcal{N}) \subseteq \langle \Gamma \rangle$. From Theorem 3.6, we can build some MPA \mathcal{S} over a refined set of messages Λ' such that $\mathcal{L}_{fifo}(\mathcal{N}) = \lambda(\mathcal{L}_{fifo}(\mathcal{S}))$ for some $\lambda : \Lambda' \to \Lambda$. Let Γ' be the subset of FIFO basic MSCs M over Λ' such that $\lambda(M) \in \Gamma$. Then $\mathcal{L}_{fifo}(\mathcal{N}) = \lambda(\mathcal{L}_{fifo}(\mathcal{S}) \cap \langle \Gamma' \rangle)$. Since $\mathcal{L}_{fifo}(\mathcal{S}) \cap \langle \Gamma' \rangle$ is recognizable and finitely generated, it is described by some globally cooperative HMSC [16, Th. 2.3].

In [19, Prop. 7], it was shown that it is undecidable whether the language of some given MPA is finitely generated. Since the language of any MPA is also the language of some netchart that we can effectively build (Th. 3.7), we obtain easily a first corollary of Th. 4.3.

Corollary 4.4. Given some netchart \mathbb{N} , it is undecidable whether $\mathcal{L}_{fifo}(\mathbb{N})$ is described by some HMSC.

Thus, it is undecidable whether a netchart language is rational. In the end of this section we show that the opposite question is undecidable, too (Th. 4.7).

4.3 From HMSCs to Netcharts

Let us now relate the notions of regularity and channel-boundedness in the framework of netcharts. Recall first that the channel-width of some basic MSC

M is the maximal number of messages that may be sent in a channel but not received along some linear extension of M. Formally, the *channel-width* of M is

 $\max_{(i,j,x)\in\mathcal{K}}\{|v|_{i!^{x}j}-|v|_{j?^{x}i}\mid u\in \mathrm{LE}(M)\wedge v \text{ is a prefix of } u\}.$

A language of basic MSCs $\mathcal{L} \subseteq \mathbf{bMSC}$ is called *channel-bounded* by an integer *B* if each basic MSC of \mathcal{L} has a channel-width at most *B*. It was observed in [8] that each regular MSC language is channel-bounded. In general the converse fails. However, for netchart languages the two notions coincide as the next elementary observation shows.

Lemma 4.5. Let \mathcal{N} be a netchart. The language $\mathcal{L}_{fifo}(\mathcal{N})$ is regular iff it is channel-bounded.

This result may be seen as a direct consequence of Theorem 3.6 although it is much easier to prove it directly. With the help of Lemma 4.5 and Th. 3.5 we can now easily characterize which channel-bounded FIFO HMSCs describe a netchart language.

Theorem 4.6. Let \mathcal{H} be a HMSC such that $\mathcal{L}_{\mathcal{H}}$ is channel-bounded and FIFO. Then $\mathcal{L}_{\mathcal{H}}$ is regular iff $\mathcal{L}_{\mathcal{H}}$ is a netchart language.

By means of the proof technique of [8, Th. 4.6], we can show easily that it is undecidable whether a channel-bounded FIFO HMSC describes a regular language. As a consequence, we get the following negative result.

Theorem 4.7. It is undecidable whether the language of some given HMSC can be described by some netchart. This holds even if we restrict to HMSCs that describe channel-bounded languages.

5 Two Positive Results for FIFO Netcharts

We have proved in Cor. 3.8 that checking regularity of $\mathcal{L}_{fifo}(\mathcal{N})$ is undecidable. To cope with this negative result, we introduce a subclass of netcharts for which regularity becomes decidable. This restriction was also considered at some point in [17].

Definition 5.1. A netchart \mathbb{N} is called FIFO if any execution sequence of its low-level Petri net is a linear extension of some FIFO basic MSC.

Figure 7 shows a non-FIFO netchart whereas Figure 4 shows a FIFO netchart. Interestingly, this subclass of netcharts is decidable and regularity is decidable in this subclass.

Theorem 5.2. It is decidable whether a netchart is a FIFO netchart.

Proof. We consider two distinct messages a and b from Λ° . These two messages are involved in four transitions $i!^a j$, $j?^a i$, $i!^b j$ and $j?^b i$ in the low-level net $\mathcal{P}_{\mathcal{N}}$. In order to check whether b can overtake a in some execution sequence of $\mathcal{P}_{\mathcal{N}}$,



Fig. 8. Construction to decide whether some netchart is FIFO

we build a new Petri net from $\mathcal{P}_{\mathcal{N}}$ by adding some places and some transitions. More precisely, around the four transitions related to a and b and the two corresponding places depicted in gray in Fig. 8, we add 8 new transitions $i!^{a_k}j$, $j?^{a_k}i$, $i!^{b_k}j$ and $j?^{b_k}i$ ($k \in \{1,2\}$) and 18 new places drawn in black in Fig. 8. Observe that the new transition $i!^{a_1}j$ can be executed at most once; moreover in this case a token is put in the new place at its left. A similar observation holds for $j?^{a_1}i$, $i!^{b_1}j$, and $j?^{b_1}i$. Observe also that $i!^{b_1}j$ can be executed only after $i!^{a_1}j$ whereas $j?^{a_1}i$ can be executed only after $j?^{b_1}i$. Now each arc from a place p to the transition $i!^{a_j}j$ is copied into an arc from p to $i!^{a_1}j$ and another arc from p to $i!^{a_2}j$. We proceed similarly with places in $i!^aj^{\circ}$ and with the transition $i!^{b_j}$. Now we claim that some MSC of $\mathcal{P}_{\mathcal{N}}$ shows some overtaking of b over a iff the new Petri net admits an execution sequence that involves the transitions $i!^{a_1}j$ and $i!^{b_1}j$. We can check the existence of such an execution sequence by reachability analysis [15].

Theorem 5.3. Regularity of $\mathcal{L}_{fifo}(\mathbb{N})$ is decidable for FIFO netcharts.

Proof. By Lemma 4.5, we have to check whether $\mathcal{L}_{flfo}(\mathcal{N})$ is channel-bounded. Since \mathcal{N} has finitely many final states, we may assume that \mathcal{N} has a unique final marking. Since $\mathcal{L}_{flfo}(\mathcal{N}) = \pi^{\circ}(\mathcal{L}_{flfo}(\mathcal{P}_{\mathcal{N}}))$, $\mathcal{L}_{flfo}(\mathcal{N})$ is channel-bounded iff $\mathcal{L}_{flfo}(\mathcal{P}_{\mathcal{N}})$ is channel-bounded. Moreover $\mathcal{L}_{flfo}(\mathcal{P}_{\mathcal{N}})$ is channel-bounded iff it is regular. Since \mathcal{N} is FIFO, this holds iff the set of all execution sequences of $\mathcal{P}_{\mathcal{N}}$ is regular. This question is decidable as shown by Lambert [12, Th. 5.2]. An alternative to this proof is to apply a recent and independent work by Wimmel [21]which is also based on [12].

6 Getting Rid of the FIFO Restriction

In this section we introduce an extended semantics for netcharts which includes non-FIFO MSCs. We show that most results in the FIFO semantics remain valid with this new approach. However we exhibit a netchart that is not implementable (Ex. 6.5).

6.1 Non-FIFO Behaviors of Netcharts

Let \mathcal{N} be a netchart and $\mathcal{P}_{\mathcal{N}}$ be its low-level Petri net. The non-FIFO language $\mathcal{L}(\mathcal{P}_{\mathcal{N}})$ of $\mathcal{P}_{\mathcal{N}}$ consists of the (possibly non-FIFO) basic MSCs M such that each

linear extension from LE(M) is an execution sequence of $\mathcal{P}_{\mathcal{N}}$. In particular, $\mathcal{L}_{fifo}(\mathcal{P}_{\mathcal{N}})$ consists of all FIFO basic MSCs of $\mathcal{L}(\mathcal{P}_{\mathcal{N}})$. When dealing with non-FIFO basic MSCs and labellings, one has to take care of degenerating MSCs.

Definition 6.1. Let Λ_1 and Λ_2 be two sets of messages and $\lambda : \Lambda_1 \to \Lambda_2$ be a mapping from Λ_1 to Λ_2 . A basic MSC $M = (E, \preccurlyeq, \xi)$ over Λ_1 is called degenerating with λ if the dag $\lambda(M) = (E, \preccurlyeq, \lambda \circ \xi)$ is not the MSC dag of some basic MSC.

Example 6.2. Consider the drawings of Fig. 9. The directed acyclic graph D' is obtained from the MSC dag D with the labelling π° such that $a_1, a_2 \mapsto a$ and $b_1 \mapsto b$. Since D' is not an MSC dag, the basic MSC D is degenerating with π° .

Since we do not want to deal with degenerate behaviors in this paper, we have to select from the basic MSCs of the low-level Petri net only those basic MSCs that are not degenerating with the labelling π° .

Definition 6.3. The non-FIFO semantics $\mathcal{L}(\mathcal{N})$ of a netchart \mathcal{N} consists of the basic MSCs obtained from the basic MSCs of $\mathcal{L}(\mathcal{P}_{\mathcal{N}})$ that are not degenerating with π° :

 $\mathcal{L}(\mathcal{N}) = \{\pi^{\circ}(M) \mid M \in \mathcal{L}(\mathcal{P}_{\mathcal{N}}) \land M \text{ is not degenerating with } \pi^{\circ}\}.$

Example 6.4. Consider the netchart \mathcal{N}_2 of Fig. 9 for which a marking \mathfrak{m} is final if $\sum_{\operatorname{Ins}(p)=i} \mathfrak{m}(p) = 1$ for each instance $i \in \mathcal{I}$. As explained in Example 6.2 the basic MSC $D \in \mathcal{L}(\mathcal{P}_{\mathcal{N}_2})$ is degenerating with π° .



Fig. 9. Netchart \mathcal{N}_2 and a degenerate behavior D' $(D' \notin \mathcal{L}(\mathcal{N}_2))$

6.2 Non-FIFO Semantics of MPAs

A rather natural non-FIFO semantics for MPAs and a corresponding notion of implementation may be defined as follows. First, the non-FIFO semantics $\mathcal{L}(S)$ of an MPA S consists of the (possibly non-FIFO) basic MSCs M such that each linear extension of M is an execution sequence of S. Now, an MSC language \mathcal{L} is *implementable under the non-FIFO semantics of MPAs* if there are some MPA S over some set of messages Λ' and some labelling $\lambda : \Lambda' \to \Lambda$ such that no MSC from $\mathcal{L}(S)$ is degenerating with λ and $\mathcal{L} = \lambda(\mathcal{L}(S))$. Differently from the FIFO semantics, *there are netcharts that are not implementable under the non-FIFO semantics*.

Example 6.5. Continuing Example 6.4, the low-level Petri net of the netchart \mathcal{N}_2 depicted in Fig. 9 admits some non-FIFO executions. However all these basic MSCs are degenerating with π° : Therefore the non-FIFO semantics of \mathcal{N}_2 consists actually of FIFO basic MSCs only. More precisely, $\mathcal{L}(\mathcal{N}_2) = \mathcal{L}_{\rm fifo}(\mathcal{N}_2)$ is described by the HMSC $(A + B)^*$ of Example 4.2. It is easy to show that *this MSC language is not implementable under the non-FIFO semantics of MPAs*.

6.3 Extending Some Results From the FIFO to the Non-FIFO Semantics

Theorems 3.7, 4.3, 4.6 and 4.7 can be established with the non-FIFO semantics by adapting the proofs slightly. Yet Corollaries 3.8 and 4.4 need to be more careful.

Theorem 6.6. It is undecidable whether some netchart language $\mathcal{L}(\mathcal{N})$ is regular (resp. can be described by some HMSC).

Proof. The proof is based on the following key technical result: For any MPA S over Λ° and any mapping $\lambda : \Lambda^{\circ} \to \Lambda$ we can effectively build a netchart \mathcal{N} such that $\mathcal{L}(\mathcal{N}) = \lambda(\mathcal{L}_{\lambda})$ where \mathcal{L}_{λ} be the set of basic MSCs $M \in \mathcal{L}(S)$ that are not degenerating with λ . Now we apply again [19, Prop. 7]. Let S be some MPA over Λ . We consider $\Lambda_{\#} = \{\#\}$ and $\lambda : \Lambda \to \{\#\}$. By the above construction, we can build some netchart \mathcal{N} such that $\mathcal{L}(\mathcal{N}) = \lambda(\mathcal{L}_{\text{fifo}}(S))$ because $\mathcal{L}_{\lambda} = \mathcal{L}_{\text{fifo}}(S)$. Then $\mathcal{L}(\mathcal{N})$ is finitely generated (resp. regular) iff $\mathcal{L}_{\text{fifo}}(S)$ is also finitely generated (resp. regular).

Discussion. These undecidability results rely essentially on the possible presence of degenerating MSCs in the low-level Petri net. Similarly to results obtained for FIFO netcharts (Th. 5.2 and 5.3), we can check effectively whether a netchart admits some degenerating MSCs in its low-level Petri net. Moreover, in case no such MSC appears, then $\mathcal{L}(N)$ is easily implementable under the non-FIFO semantics of MPAs and we can effectively check whether it is regular. Thus, it is quite useful to avoid degenerate behaviors. For this reason, we suggest that component MSCs should use disjoint set of messages (that is, messages should be private to transitions) because this simple re-

quirement ensures that no degenerating MSC appears in the low-level Petri net.

Acknowledgements. Thanks to the anonymous referees for suggestions to improve the presentation of the paper. We thank also H. Wimmel for communicating us paper [21].

References

- Alur R., Etessami K. and Yannakakis M.: Realizability and verification of MSC graphs. ICALP, LNCS 2076 (2001) 797–808
- Alur R. and Yannakakis M.: Model Checking of Message Sequence Charts. CON-CUR, LNCS 1664 (1999) 114–129
- 3. Baudru N. and Morin R.: Safe Implementability of Regular Message Sequence Charts Specifications. Proc. of the ACIS 4th Int. Conf. SNDP (2003) 210–217
- Bollig B., Leucker M. and Noll Th.: Generalised Regular MSC Languages. FoSSaCS, LNCS 2303 (2002) 52–66
- 5. Caillaud B., Darondeau Ph., Hélouët L. and Lesventes G.: *HMSCs as partial spec-ifications... with PNs as completions.* LNCS **2067** (2001) 87–103
- 6. Diekert V. and Rozenberg G.: The Book of Traces. (World Scientific, 1995)
- 7. Gunter E.L., Muscholl A. and Peled D.: *Compositional Message Sequence Charts*. TACAS, LNCS **2031** (2001) 496–511
- Henriksen J.G., Mukund M., Narayan Kumar K. and Thiagarajan P.S.: On message sequence graphs and finitely generated regular MSC language. ICALP, LNCS 1853 (2000) 675–686
- 9. Henriksen J.G., Mukund M., Narayan Kumar K. and Thiagarajan P.S.: Regular collections of message sequence charts. MFCS, LNCS 1893 (2000) 405–414
- 10. Holzmann G.J.: Early Fault Detection. TACAS, LNCS 1055 (1996) 1-13
- 11. ITU-TS: Recommendation Z.120: Message Sequence Charts. (Geneva, 1996)
- 12. Lambert J.L.: A structure to decide reachability in Petri nets. Theoretical Comp. Science **99** (1992) 79–104
- 13. Lamport L.: *Time, Clocks and the Ordering of Events in a Distributed System.* Communications of the ACM **21,7** (1978) 558–565
- 14. Lohrey M.: *Realizability of High-level Message Sequence Charts: closing the gaps.* Theoretical Comp. Science **309** (2003) 529–554
- 15. Mayr E.W.: An algorithm for the general Petri net reachability problem. SIAM Journal of Computing 13:3 (1984) 441–460
- Morin R.: Recognizable Sets of Message Sequence Charts. STACS 2002, LNCS 2285 (2002) 523–534
- Mukund M., Narayan Kumar K. and Thiagarajan P.S: Netcharts: Bridging the Gap between HMSCs and Executable Specifications. CONCUR 2003, LNCS 2761 (2003) 296–310
- 18. Muscholl A. and Peled D.: Message sequence graphs and decision problems on Mazurkiewicz traces. MFCS, LNCS 1672 (1999) 81–91
- 19. Muscholl A. and Peled D.: From Finite State Communication Protocols to Highlevel Message Sequence Charts. ICALP, LNCS 2076 (2001) 720–731
- Pratt V.: Modelling concurrency with partial orders. International Journal of Parallel Programming 15 (1986) 33–71
- 21. Wimmel H.: Infinity of Intermediate States is Decidable for Petri Nets. Applications and Theory of Petri Nets, LNCS (2004) –To appear

Basic Theory of Reduction Congruence for Two Timed Asynchronous π -Calculi

Martin Berger

Dept. of Computer Science, Queen Mary, Univ. of London

Abstract. We study reduction congruence, the most widely used notion of equality for the asynchronous π -calculus with timers, and derive several alternative characterisations, one of them being a labelled asynchronous bisimilarity. These results are adapted to an asynchronous π -calculus with timers, locations and message failure. In addition we investigate the problem of how to distribute value-passing processes in a semantics-preserving way.

1 Introduction

The π -calculus has been used to good effect as a tool for modelling and reasoning about computation [6,7,18,23,26]. Unfortunately, it appears incomplete for compositional representation and verification of distributed systems. An important instance of what cannot be covered convincingly are network protocols, for example TCP, that implement reliable (under some mild constraints about the probability of message failures) FIFO channels on top of an unreliable message passing fabric. Typically, such protocols start a timer when sending a message and, if the corresponding acknowledgement doesn't arrive early enough or not at all, a time-out initiates a retransmission. Timed Automata, Time(d) Petri Nets, Timed CCS and many other formalisms have been proposed to help express this or similar phenomena. Unfortunately, they all seem insufficient to give convincing accounts of advanced programming languages containing primitives for distribution, such as Java or the POSIX libraries. The two key shortcomings are the lack in expressivity of the underlying non-distributed formalism (e.g. finite automata or CCS do not allow precise and compositional modelling of Java's non-distributed core) and incomplete integration of the different features that are believed to be necessary for modelling distributed systems (e.g. [1] lacks timing and many timed process algebras do not feature message failures among their primitive operations). As an initial move towards overcoming this expressivity gap, [5] augmented the asynchronous π -calculus with a timer, with locations, message-loss, location failure and the ability to save process state. The present text, a partial summary of [4], takes the next step and starts the study of two extensions in earnest by investigating the natural equality for π_t , the asynchronous π -calculus with timers and π_{mlt} , the asynchronous π -calculus with timers and message failure.

The remainder has two main parts. First, several characterisations of π_t 's reduction congruence, the canonical equivalence for asynchronous π -calculi [12,

[©] Springer-Verlag Berlin Heidelberg 2004

17], are obtained. The most useful of those is as a labelled bisimilarity. For other untyped π -calculi, weak labelled characterisations of reduction congruence have not been forthcoming, only sound approximations. π_t is interesting because it allows to study the effect of combining discrete timing and name passing interaction, a line of inquiry long overdue. It also paves the way for the second part which studies π_{mlt} , a minimal extension of π_t allowing convenient expression of basic distributed algorithms such as the aforementioned network protocol. We show that reasoning about π_{mlt} can be broken down into two parts: reasoning about processes, i.e. reasoning in π_t , and reasoning about distributed interaction. A related aim is to devise a translation $(\cdot)^{\bullet}$ that allows to take a non-distributed process P | Q and locate it as $[P^{\bullet}] | [Q^{\bullet}]$ in a semantics preserving way (here $[\bullet]$ denotes location). This may help to reason about some properties of distributed processes using tools from centralised computing. That may not be possible for arbitrary P and Q but we identify a translation that works for a restricted class of processes and may be a good starting point for further investigations. The other main contribution of this second part is a characterisation of π_{mlt} 's reduction congruence by a barbed congruence [21], and a sound approximation by a labelled bisimilarity.

2 Adding Discrete Timing to π

2.1 Syntax and Semantics of π_t

The π -calculus [16,20] is a simple syntax for modelling computation as namepassing interaction. The key operational rule is

$$(COM) \quad \overline{x}\langle \tilde{y}\rangle \,|\, x(\tilde{v}).\mathsf{P} \to \mathsf{P}\{\tilde{y}/\tilde{v}\}$$

where the process $\overline{x}\langle \tilde{y} \rangle$ sends the data \tilde{y} along a channel x (drawn from a countably infinite set \mathcal{N} of *names*) and another process $x(\tilde{v})$. P waits to receive data on the same channel x, called *input subject*. When the interaction happens, $x(\tilde{v})$. P evolves into $P\{\tilde{y}/\tilde{v}\}$. The operator | is *parallel composition*. Finitely describable infinitary behaviour is achieved by way of a *replication* operation ! and interaction of P at x with the environment can be prevented by the *restriction* (νx)P.

Our new syntax timer^t($x(\tilde{v})$, P, Q), with t > 0 being an integer, is straightforward and completely standard. It supports two operations: (1) the *time-out* which means that after t steps it turns into Q, unless (2) it has been *stopped*, i.e. that a message has been received by the timer at x.

(STOP) timer^{t+1}(
$$x(\tilde{v})$$
, P, Q) | $\overline{x}\langle \tilde{y} \rangle \to \mathsf{P}\{\tilde{y}/\tilde{v}\}$

The resulting calculus is given by the following grammar.

$$\mathsf{P} ::= x(\tilde{y}).\mathsf{P} \mid \overline{x}\langle \tilde{y} \rangle \mid \mathsf{P}|\mathsf{Q} \mid (\nu x)\mathsf{P} \mid \mathsf{timer}^t(x(\tilde{v}).\mathsf{P},\mathsf{Q}) \mid !x(\tilde{v}).\mathsf{P} \mid \mathsf{0}$$

We often abbreviate $\overline{x}\langle\rangle$ to \overline{x} and write x.P in x().P's stead. The asynchronous π -calculus is a sub-calculus of π_t . The free and bound names of timers are:

fn(timer^t($x(\tilde{v})$.P,Q)) = fn($x(\tilde{v})$.P) \cup fn(Q), bn(timer^t($x(\tilde{v})$.P,Q)) = bn($x(\tilde{v})$.P) \cup bn(Q). Structural congruence \equiv is defined by the same axioms as in the asynchronous π -calculus, but over our extended syntax.

The flow of time is communicated at each step in the computation by a *time stepper function* ϕ , which acts on processes. It models the implicit broadcast of time passing and works equally well for labelled transitions and reductions.

$$\phi(\mathsf{P}) = \begin{cases} \mathsf{timer}^{t-1}(x(\tilde{v}).\mathsf{Q},\mathsf{R}) & \mathsf{P} = \mathsf{timer}^t(x(\tilde{v}).\mathsf{Q},\mathsf{R}), t > 1\\ \mathsf{R} & \mathsf{P} = \mathsf{timer}^t(x(\tilde{v}).\mathsf{Q},\mathsf{R}), t \leq 1\\ \phi(\mathsf{Q})|\phi(\mathsf{R}) & \mathsf{P} = \mathsf{Q}|\mathsf{R}\\ (\nu x)\phi(\mathsf{Q}) & \mathsf{P} = (\nu x)\mathsf{Q}\\ \mathsf{P} & \text{otherwise.} \end{cases}$$

Here is how time stepping is used.

(PAR)
$$P \rightarrow P' \Rightarrow P|Q \rightarrow P'|\phi(Q)$$

The only difference with the corresponding rule of untimed calculi is that we have $\phi(\mathbf{Q})$ rather than \mathbf{Q} in the resulting process of the conclusion. It ensures that each *active timer*, that is any timer not under a prefix, is ticked one unit at each interaction. The additional rule

(IDLE) $\mathsf{P} \to \phi(\mathsf{P})$

prevents the flow of time from ever being halted by deadlocked processes. This means, π_t does not enforce progress assumptions that can be found in many models of timed computations. It is possible to add progress requirements later on top of π_t . Here are the remaining reduction rules.

$$\begin{array}{ccc} (\text{Rep}) & !x(\tilde{v}).\mathsf{P} \mid \overline{x}\langle \tilde{y} \rangle & \rightarrow !x(\tilde{v}).\mathsf{P} \mid \mathsf{P}\{\tilde{y}/\tilde{v}\} \\ (\text{Res}) & \mathsf{P} \to \mathsf{Q} \Rightarrow (\nu x)\mathsf{P} \to (\nu x)\mathsf{Q} & (\text{Cong}) & \mathsf{P} \equiv \mathsf{P}' \to \mathsf{Q}' \equiv \mathsf{Q} \Rightarrow \mathsf{P} \to \mathsf{Q} \end{array}$$

The corresponding labelled synchronous semantics is obtained from the conventional synchronous semantics [15] of the asynchronous π -calculus with the following new rules, the first of which replacing that for parallel composition.

$$P \xrightarrow{l} P', bn(l) \cap fn(Q) = \emptyset \implies P | Q \xrightarrow{l} P' | \phi(Q)$$

timer^{t+1}(x(\tilde{v}).P, Q) $\xrightarrow{x(\tilde{z})} P\{\tilde{z}/\tilde{v}\}$ $P \xrightarrow{\tau} \phi(P)$

Labels are given as $l ::= \tau | x(\tilde{y}) | \overline{x} \langle (\nu \tilde{y}) \tilde{z} \rangle$. Contexts are standard except for two new rules: $C[\cdot] ::= ... | \operatorname{timer}^t(x(\tilde{v}).\mathsf{P}, C'[\cdot]) | \operatorname{timer}^t(x(\tilde{v}).C'[\cdot], \mathsf{P})$. A binary relation \mathcal{R} on processes is a π_t -congruence if it is an equivalence, if $\equiv \subseteq \mathcal{R}$ and if $\mathsf{P} \ \mathcal{R} \ \mathsf{Q}$ implies $C[\mathsf{P}] \ \mathcal{R} \ C[\mathsf{Q}]$ for all contexts $C[\cdot]$. The strong barb \downarrow_x for π_t is defined (up to \equiv) by (1) $\overline{x} \langle \tilde{y} \rangle \downarrow_x$; (2) $\mathsf{P} \downarrow_x \Rightarrow \mathsf{P} | \mathsf{Q} \downarrow_x$; and (3) $\mathsf{P} \downarrow_x$, $x \neq y \Rightarrow (\nu y) \mathsf{P} \downarrow_x$. A symmetric binary relation \mathcal{R} on processes is a strong barbed bisimulation if it is a π_t -congruence and if $\mathsf{P} \ \mathcal{R} \ \mathsf{Q}$ implies the following: (1) for all names $x: \mathsf{P} \downarrow_x$ implies $\mathsf{Q} \downarrow_x$; and (2) whenever $\mathsf{P} \to \mathsf{P}'$ then there is a process Q' such that $\mathsf{Q} \to \mathsf{Q}'$ and $\mathsf{P}' \mathcal{R} \mathsf{Q}'$. The largest strong barbed bisimulation $\stackrel{rc}{\sim}$ is strong reduction congruence. The corresponding notions of barbed bisimulation and reduction congruence $\stackrel{rc}{\approx}$ are derived by replacing \downarrow_x with \Downarrow_x and \to with \to . Here \to is the transitive and reflexive closure of \to and $P \Downarrow_x$ means $P \to Q \downarrow_x$ for some Q. A binary relation \mathcal{R} on processes is *time-closed* if $\mathsf{P} \mathcal{R} \mathsf{Q}$ implies $\phi(\mathsf{P}) \mathcal{R} \phi(\mathsf{Q})$. It will later emerge that $\stackrel{rc}{\approx}$ and $\stackrel{rc}{\sim}$ are time-closed.

Examples (1)

- 1. The process $delay^t(P) = (\nu x)timer^t(x.0, P)$ implements a *delay operator*, assuming $x \notin fn(P)$. For t units of time, it cannot interact at all, it behaves like 0, but then it evolves into P. It is comparable to the sleep operator in Java and can be used to implement cyclic behaviour: $(\nu x)(\overline{x} \mid !x.delay^t(P \mid \overline{x}))$ $(x \notin fn(P))$ which spawns P every t + 1 units of time. The delay operator is crucial in the proof of Theorem 2.
- 2. The next example shows that we only need timer¹($x(\tilde{v})$.P,Q) as timing construct. As all others can be built up by iteration of this basic form. Define $T_1 = timer^1(x(\tilde{v})$.P,Q) and $T_{t+1} = timer^1(x(\tilde{v})$.P, $T_t)$. Then $T_t \stackrel{rc}{\sim} timer^t(x(\tilde{v})$.P,Q) for all t > 0.
- 3. Assume that P is a process of the form $x(\tilde{v}).Q$. Define P⁰ to be 0 and let $P^{n+1} = timer^1(P, P^n)$. Then Pⁿ is a process that offers the service P for n time units when it becomes unguarded. Note that P is offered only once. delayⁿ(P^m) also offers P for n units of time, but not straight away. Instead the service is available only after m units of time.
- 4. A variant of the previous example. Assume P = x(v).Q with fn(P) = {x}, x ∉ {v}. Let P⁰ = 0 and set Pⁿ⁺¹ = timer¹(x(v).(Q | Pⁿ), Pⁿ). Now P is offered for repeated use in Pⁿ, for n units of time, so we may invoke P up to n times.

2.2 Why a Novel Kind of Timer?

Before getting on with the technical development, we'd like to summarise the key reasons for devising our own reduction-based account of discrete timing rather than adapting one of the existing constructs.

- A key design objective was simplicity and preservation of as much established π -calculus technology as possible. That ruled out labelled transitions with dedicated time passing actions to communicate the flow of time. The ability to use the simpler reduction semantics is advantageous because it is sometimes difficult to find suitable labelled semantics. It is trivial to adapt the timer proposed here to other models of computing, from Ambient Calculi [11], to λ -calculi and Abstract State Machines [9]. This is currently not possible for labelled-transition based approaches to timing.
- Some previous proposals exhibit behavioural anomalies, such as timers being able to stop themselves. This is caused, to put it simplistically, by less than ideal combinations of progress assumptions, the ability for time to pass under

unrestricted sums and computational steps having zero duration. The calculi proposed here do not suffer from these shortcomings.

- Finally, we must emphasise that our timer is different from those where timeflow is communicated by labelled transitions only in its syntactic presentation. Its behaviour is essentially identical. Semantically relevant differences between our calculus and its alternatives are a consequence of other design choices, for example progress assumptions or the presence of mixed choice, not of the presentation of timers by way of time-steppers.

Our design of π_t and π_{mlt} is discussed in great detail in [4], which also contains comparisons with the alternative approaches.

2.3 The Maximal Sound Theory

Reduction congruence is often seen to be the most canonical equivalence for asynchronous π -calculi. This section looks at its incarnation for π_t . The presentation is close to [17] to facilitate comparison, but due to timers, proofs are quite different.

A logic is a pair $\mathcal{L} = (F, \vdash)$ comprising a set F of formulae and an entailment relation $\vdash \subseteq \mathcal{P}(F) \times F$. In this section, F will always be pairs of π_t -processes. References to the underlying logic \mathcal{L} will often be omitted. A set T of formulae is a π_t -theory, or simply a theory, and its members are axioms. We write $T \vdash P = Q$ whenever $(T, (P, Q)) \in \vdash$ and call (P, Q) a theorem or consequence of T in \mathcal{L} . If $T \vdash P = Q$ is not derivable, we write $T \nvDash P = Q$. The set of all consequences of T in \mathcal{L} is denoted $|T|_{\mathcal{L}}$ (with the subscript \mathcal{L} often omitted). T is consistent if |T| does not equate all processes, otherwise it is inconsistent. T is reductionclosed if $T \vdash P = Q$ and $P \twoheadrightarrow P'$ implies the existence of a reduction sequence $Q \twoheadrightarrow Q'$ such that $T \vdash P' = Q'$. T is strongly reduction-closed if $T \vdash P = Q$ and $P \to P'$ implies the existence of a reduction $Q \to Q'$ such that $T \vdash P' = Q'$. In this section we only use π_t -logics (T, \vdash) whose entailment is inductively defined such that |T| is a π_t -congruence containing T. T is time-closed if $T \vdash P = Q$ implies $T \vdash \phi(P) = \phi(Q)$.

As is well-known, there is no unique largest consistent and reduction-closed theory (Theorem 1.2 below), so we have to impose a mild additional constraint. Preservation of weak barbs is a popular choice, but requires a notion of observation. Alas, it is not apriori clear what observing timed computations may entail. Fortunately, we can do without a notion of observation and will *prove* in Theorem 1 that \downarrow_x defined above is in fact a correct notion of barb. A process P is *insensitive* if it can never interact with any other process, i.e. P $\rightarrow Q$ implies $an(Q) = \emptyset$. Here an(P), the *active names* of P, is given by induction on the syntax of P: $an((\nu x)P) = an(P) \setminus \{x\}, an(P|Q) = an(P) \cup an(Q), an(0) = \emptyset$ and $an(x(\tilde{v}).P) = an(!x(\tilde{v}).P) = an(timer^t(x(\tilde{v}).P,Q)) = an(\overline{x}(\tilde{y})) = \{x\}$. A π_t theory is *sound* if it is consistent, reduction-closed and equates any two insensitive terms.

The dramatic semantic effect of timers becomes apparent in the next proposition: we are guaranteed *strong* reduction-closure despite having stipulated only reduction-closure. **Proposition 1.** Let T be sound. (1) If $T \vdash P = Q$, then: $P \downarrow_x$ if and only if $Q \downarrow_x$. (2) If $T \vdash P = Q$ then for all appropriate \tilde{x} , \tilde{v} : $T \vdash P\{\tilde{x}/\tilde{v}\} = Q\{\tilde{x}/\tilde{v}\}$. (3) If T is a sound theory, then T is time-closed. (4) T is reduction-closed if and only if, whenever $T \vdash P = Q$, then, for all contexts $C[\cdot], C[P] \rightarrow P'$ implies $C[Q] \rightarrow Q'$, for some Q' with $T \vdash P' = Q'$.

The key reason why requiring reduction-closure and congruency gives strong reduction-closure is (roughly) that we can use a process like timer¹($x(\tilde{v}).\bar{a}, 0$) to detect and signal the fact that $P \downarrow_x$ by running both in parallel. After the first step of the clock, that ability disappears forever. Hence any process that wishes to be equated to P by a sound theory better be able to match any of P's strong barbs immediately and not only after some reduction steps.

With $\mathcal{T}_{\max} = \bigcup \{ \mathcal{T} \mid \mathcal{T} \text{ is a sound theory} \}$ we can now state the existence and various alternative presentations of the maximal sound theory.

Theorem 1. (1) T_{max} is the unique sound theory such that $|T| \subseteq |T_{max}|$ for all sound theories T. T_{max} is called the maximum sound theory. (2) There is no largest consistent, reduction-closed theory. (3) $|T_{max}| = T_{max} = \stackrel{rc}{\approx} = \stackrel{rc}{\approx}$.

2.4 Labelled Semantics

Reduction based equivalences are sometimes hard to use. To make reasoning easier, labelled semantics and associated notions of bisimilarities have been developed for many untimed calculi. We shall now do the same for π_t . A symmetric binary relation \mathcal{R} is a *strong synchronous bisimulation* if $\mathsf{P} \ \mathcal{R} \ \mathsf{Q}$ and $\mathsf{P} \stackrel{l}{\longrightarrow} \mathsf{P}'$ means that there is a synchronous transition $\mathsf{Q} \stackrel{l}{\longrightarrow} \mathsf{Q}'$ with $\mathsf{P}' \ \mathcal{R} \ \mathsf{Q}'$. The largest strong synchronous bisimulation \sim is *strong synchronous bisimilarity*. Weak bisimilarity \approx is defined by replacing $\mathsf{Q} \stackrel{l}{\longrightarrow} \mathsf{Q}'$ with $\mathsf{Q} \stackrel{\hat{l}}{\longrightarrow} \mathsf{Q}'$ ($\hat{\cdot}$ is the usual τ -erasing operation).

The failure of the various synchronous bisimilarities to equate \mathbf{fw}_{xx} with 0 has lead to asynchronous transitions [15] which model asynchronous observers. Since \mathcal{T}_{max} , unlike \approx and \sim , equates \mathbf{fw}_{xx} and 0, asynchronous bisimilarity might also be interesting in π_t (here $\mathbf{fw}_{xy} = !x(\tilde{v}).\bar{y}\langle \tilde{v} \rangle$). But what are asynchronous transitions $\stackrel{l}{\longrightarrow}_a$? Unfortunately, the straightforward adaptation to π_t of the transitions introduced in [15] does not work, because the obvious rule for parallel composition

$$\mathsf{P} \xrightarrow{l} {a} \mathsf{P}', \mathsf{bn}(l) \cap \mathsf{fn}(\mathsf{Q}) = \emptyset \quad \Rightarrow \quad \mathsf{P} | \mathsf{Q} \xrightarrow{l} {a} \mathsf{P}' | \phi(\mathsf{Q}) \tag{1}$$

does not connect asynchrony well with time passing. To see what goes wrong consider what it means to be an asynchronous observer. Interacting with a process to detect that it sends a message consumes one unit of time. The (PAR) rule and its labelled counterpart (1) ensure that this time-step permeates all processes. Dually, testing that a process is inputting involves sending a message. But asynchronously entails that the observer cannot know exactly when the message has been consumed. Hence the observation $\frac{x(\tilde{v})}{a}$ should not be associated with

a time step, for otherwise a judiciously set timer could detect that interaction by the time it takes. So the rule (1) for parallel composition above may work incorrectly. We propose to split it in two:

$$- P \xrightarrow{\iota}_{a} P', \ l \neq x(\tilde{v}), \ bn(l) \cap fn(Q) = \emptyset \Rightarrow P \mid Q \xrightarrow{\iota}_{a} P' \mid \phi(Q)$$
$$- P \xrightarrow{x(\tilde{y})}_{a} P', \ bn(l) \cap fn(Q) = \emptyset \Rightarrow P \mid Q \xrightarrow{x(\tilde{y})}_{a} P' \mid Q$$

The remaining rules for the inductive definition of $\frac{l}{d}a$ are here:

$$\begin{split} \overline{x}\langle \tilde{y}\rangle &\xrightarrow{\overline{x}\langle \tilde{y}\rangle}{a} 0 \qquad \mathsf{P} \xrightarrow{l}{a} \mathsf{Q}, x \notin \mathsf{fn}(l) \cup \mathsf{bn}(l) \Rightarrow (\nu x)\mathsf{P} \xrightarrow{l}{a} (\nu x)\mathsf{Q} \\ \overline{x}\langle \tilde{y}\rangle &| x(\tilde{v}).\mathsf{Q} \xrightarrow{\tau}{a} \mathsf{Q}\{\tilde{y}/\tilde{v}\} \qquad \overline{x}\langle \tilde{y}\rangle &| !x(\tilde{v}).\mathsf{Q} \xrightarrow{\tau}{a} \mathsf{Q}\{\tilde{y}/\tilde{v}\} &| !x(\tilde{v}).\mathsf{Q} \\ \overline{x}\langle \tilde{y}\rangle &| \mathsf{timer}^{t}(x(\tilde{v}).\mathsf{Q},\mathsf{R}) \xrightarrow{\tau}{a} \mathsf{Q}\{\tilde{y}/\tilde{v}\} \qquad \mathsf{P} \xrightarrow{\tau}{a} \phi(P) \\ \mathsf{P} \equiv \mathsf{P}', \mathsf{P}' \xrightarrow{l}{a} \mathsf{Q}', \mathsf{Q}' \equiv \mathsf{Q} \Rightarrow \mathsf{P} \xrightarrow{l}{a} \mathsf{Q} \qquad 0 \xrightarrow{x(\tilde{z})}{a} \overline{x}\langle \tilde{z}\rangle \\ \mathsf{P} \xrightarrow{\overline{x}(\langle \nu \tilde{y} \rangle \tilde{z})}{a} \mathsf{Q}, a \neq x, a \in \{\tilde{z}\} \setminus \{\tilde{y}\} \Rightarrow (\nu a)\mathsf{P} \xrightarrow{\overline{x}(\langle \nu \tilde{y}, a \rangle \tilde{z})}{a} \mathsf{Q} \end{split}$$

The set of labels is the same as for synchronous transitions. *Strong asynchronous bisimilarity* \sim_a and its weak counterpart \approx_a are defined just as (strong) synchronous bisimilarity except that $\stackrel{l}{\longrightarrow}$ is replaced with $\stackrel{l}{\longrightarrow}_a$. The next lemma shows that timers also wreak havoc with labelled equivalences.

Lemma 1. Neither \approx nor \sim , \approx_a and \sim_a are closed under parallel composition.

As an example of what may go wrong, note that $\mathsf{P} \stackrel{\text{def}}{=} (\nu x)(\overline{x} | \mathsf{timer}^1(x.\overline{y}, 0)) \sim (\nu x)(\overline{x} | \mathsf{timer}^1(x.0,\overline{y})) \stackrel{\text{def}}{=} \mathsf{Q}$ means $\mathsf{P} \mathcal{R} \mathsf{Q} (\mathcal{R} \text{ is any of the four equivalences in Lemma 1) but <math>\mathsf{Q} | \overline{a} \xrightarrow{\overline{a}} \xrightarrow{\overline{y}} (\nu x)\overline{x}$ cannot be matched by $\mathsf{P} | \overline{a}$.

This failure of closure under parallel composition is caused by lacking timeclosure. Let \sim'_a be the largest strong, asynchronous bisimulation that is also time-closed, with \approx' , \sim' and \approx'_a being defined similarly. Its easy to show that these four new equivalences are closed under parallel composition. Still, this does not guarantee congruency.

Proposition 2. Assume x, y, a, b are fresh and distinct names. Define

$$\begin{split} \mathsf{P} &= (\nu a)(\overline{x}\langle a \rangle \,|\, ! y(v).\overline{v}) \\ \mathsf{Q} &= (\nu a)(\overline{x}\langle a \rangle \,|\, ! y(v).\overline{v} \,|\, \mathsf{timer}^1(y(v).(\overline{v} \,|\, \mathsf{timer}^1(a.\overline{b},0)),0)). \end{split}$$

If \mathcal{R} is one of \approx , \sim , \approx_a , \sim_a , \approx' , \sim' , \approx'_a or \sim'_a , then $\mathsf{P} \mathcal{R} \mathsf{Q}$ but not $\mathsf{P}\{x/y\}\mathcal{R} \mathsf{Q}\{x/y\}$. Consequently, \mathcal{R} cannot be closed under any of the three available forms of input prefixing.

In the asynchronous π -calculus, various reasonable equivalences are congruences. That this fails for π_t hints at renaming carrying non-trivial computational content. Interestingly, our example uses nested timers. It is conceivable that prohibiting nesting of timers results in a subcalculus where the relevant equivalences are renaming-closed. The next result shows that failure of renaming-closure is the only defect \sim'_a has vis-a-vis congruency. Define \sim^c_a as the largest strong, time-closed, asynchronous bisimulation that is also renaming-closed. **Proposition** 3. \sim_a^c is the largest strong asynchronous bisimulation contained in \sim_a that is also a congruence.

The processes P and Q, defined just after Lemma 1, also show that fully abstract and compositional encodings $\llbracket \cdot \rrbracket$ of π_t into the asynchronous π -calculus are impossible, when the equivalence \mathcal{R} on the source of the encoding is one of those mentioned in Lemma 1. Otherwise we could derive $P \sim Q \Rightarrow \llbracket P \rrbracket \sim \llbracket Q \rrbracket \Rightarrow \llbracket P \rrbracket \Vert \overline{a} \rrbracket \bowtie \llbracket P \Vert \overline{a} \rrbracket \bowtie \llbracket P \Vert \overline{a} \rrbracket \bowtie \llbracket Q \Vert \overline{a} \rrbracket \Rightarrow P \vert \overline{a} \sim Q \vert \overline{a}$ (the target's equivalence \bowtie is only required to be closed under parallel composition for the encoding to be contradictory). The converse question is also interesting: can untimed subcalculi of π_t , for example the asynchronous π -calculus, be embedded? Once again the answer seems mostly negative: a translation $\llbracket \cdot \rrbracket$ from π_a into π_t is *barb-expansive* if for all P and all names x we can find an integer n > 0 such that $d(\llbracket P \rrbracket, x) \ge n \cdot d(P, x)$. Here d(P, x) is the least n such that $P \to \cdots \to Q \downarrow_x$

and ω if no such *n* exists. Then one can easily show the following. Assume the chosen π_a -equivalence equates \overline{x} with $\tau.\overline{x}$. If $\llbracket\cdot\rrbracket$ is a barb-expansive mapping from π_a into π_t , then it cannot be complete with reduction congruence being π_t 's equivalence. In particular, the syntactic inclusion of π_a into π_t cannot be fully abstract.

2.5 Characterising \mathcal{T}_{max} as \sim_a^c

In the asynchronous π -calculus, asynchronous bisimilarity soundly approximates the corresponding maximal theory, but does not characterise it, a counterexample being $\overline{x}\langle y \rangle | eq_{yz}$ and $\overline{x}\langle z \rangle | eq_{yz}$, where $eq_{yz} = fw_{yz} | fw_{zy} [17]$. The reason for their semantic equality is that eq_{yz} turns any observation on y into a weak observation on z and vice versa. There is no way for a process in the asynchronous π -calculus to detect whether a name has come via eq_{yz} or not. In π_t this is different because forwarding takes time. This leads to the following labelled characterisation of reduction congruence.

Theorem 2. $\mathcal{T}_{max} = \sim_a^c \subsetneq \sim_a' \subsetneq \sim_a \subsetneq \approx_a' \subsetneq \approx_a$. In addition $\sim \subsetneq \approx$ and $\approx \subsetneq \approx_a$.

The proof is straightforward, except for showing $\mathcal{T}_{\max} \subseteq \sim_a^c$. The key difficulty is to establish that $\mathcal{T}_{\max} \vdash \mathsf{P} = \mathsf{Q}$ and $\mathsf{P} \xrightarrow{\overline{x}\langle(\nu \hat{y})\hat{z}\rangle} \mathsf{P}'$ together imply $\mathsf{Q} \xrightarrow{\overline{x}\langle(\nu \hat{y})\hat{z}\rangle} \mathsf{Q}'$ for some Q' with $\mathcal{T}_{\max} \vdash \mathsf{P}' = \mathsf{Q}'$. Simplifying greatly, the proof uses a context like

$$C[\cdot] = [\cdot] \mid x(\tilde{v}).\Pi_{z_i \in \tilde{z}} \Pi_{j=1}^{f(i)} \overline{z_i} \langle \ldots \rangle$$

which receives a tuple of names at x and encodes at what positions in the tuple \tilde{v} a name w was received by encoding these positions through the number of uninterrupted (even by τ) outputs of w. Here $\prod_{i \in \{1,...,n\}} P_i \equiv P_1 |...| P_n$ and f is a suitable function allowing this encoding. The construction of f is delicate and omitted for brevity, but we cannot use simple functions like the identity $i \mapsto i$, because $C[\cdot]$ must be able to distinguish, for example, $\overline{x}\langle abaa \rangle$ from $\overline{x}\langle aaab \rangle$.

Both have the same number of as and bs. This is why we must code up not only how many times a name occurs in \tilde{y} but also at which positions. Using the observational capabilities of timers, we can distinguish processes that can output a fixed name n times, but not n + 1 times in an uninterrupted row from processes that can do more than n uninterrupted outputs of that name. Thus the sketched construction of $C[\cdot]$ ensures that $\mathcal{T}_{max} \vdash C[P] = C[Q]$ can only hold if Q can do *exactly* the same initial outputs as P, which is what was needed to be shown. The actual proof is more complicated and can be found in [4].

Examples (2). The next few examples show how easy it is to reason about \mathcal{T}_{max} with \sim_a^c .

- 1. The identity forwarder fw_{aa} and 0 are strongly reduction congruent. To see this, define \mathcal{R} up to \equiv by $\mathsf{fw}_{xx} \mid \Pi_i \overline{y_i} \langle \tilde{z}_i \rangle \mathcal{R} \prod_i \overline{y_i} \langle \tilde{z}_i \rangle$ whenever $\{y_i, \tilde{z}_i\} \subseteq \mathcal{N}$. Obviously \mathcal{R} is time- and renaming closed. Since all occurring processes are timer-free, idle transitions can trivially be matched. The only vaguely interesting transition $\mathsf{fw}_{xx} \mid \overline{x}\langle \tilde{a} \rangle \mid \Pi_i \overline{y_i} \langle \tilde{z}_i \rangle \xrightarrow{\tau}_a \mathsf{fw}_{xx} \mid \overline{x}\langle \tilde{a} \rangle \mid \Pi_i \overline{y_i} \langle \tilde{z}_i \rangle$ is clearly matched by the idle transition $\overline{x}\langle \tilde{a} \rangle \mid \Pi_i \overline{y_i} \langle \tilde{z}_i \rangle \xrightarrow{\tau}_a \overline{x}\langle \tilde{a} \rangle \mid \Pi_i \overline{y_i} \langle \tilde{z}_i \rangle$.
- matched by the idle transition \$\overline{x}\lambda \rangle |\Pi_i \overline{y}_i \lambda \vec{z}\rangle -\frac{\sigma}{\sigma} \overline{x}\lambda \vec{z}\rangle -\frac{\sigma}{\sigma} \vec{x}\lambda \vec{z}\rangle -\sigma} \vec{x}\lambda \vec{x}\lambda \vec{z}\rangle -\sigma} \vec{x}\lambda \vec{x}\lambda \vec{z}\lambda \vec{z}\rangle -\sigma} \vec{x}\lambda \vec{z}\lambda \vec{z}\rangle -\sigma} \vec{x}\lambda \vec{z}\lambda \vec{z}\lambda
- Parallel composition and delay operators commute, i.e. T_{max} ⊢ delay^t(P|Q) = delay^t(P) | delay^t(Q): consider R given by delay^t(P|Q) R delay^t(P) | delay^t(Q). It is again straightforward to verify that R ∪ id is a renaming-closed, time-closed, asynchronous bisimulation.

Locality. A process P is *local* no input is bound by another input, i.e. we do not allow processes like x(y).y(v).P. We denote π_t restricted to local processes by π_t^{loc} . Local processes are convenient for modelling distributed computing.

Theorem 3. All results stated so far also hold in π_t^{loc} .

3 Adding Location and Message Failure

One of the main uses of timers is to unblock computations after they became stuck due to some fault such as a lost message. This is inconvenient to model in π_t because it lacks message failures. To explore timers in a more realistic setting, this section augments π_t with locations and non-byzantine message failure, obtaining π_{mlt} .

3.1 Syntax and Semantics of π_{mlt}

Processes in π_{mlt} , called *networks* and closely related to, but not identical with [5], are parallel compositions of *messages in transit* $\overline{x}\langle \tilde{y} \rangle$ and *locations* or *sites* [P]_A which execute π_t processes. Restriction of names is also possible for networks using (vx). For simplicity P must be *local* and the subscript A contains the free

names that $[P]_A$ may use to receive data on. Messages in transit have left their source location but not yet arrived at the destination. Message failure occurs only in transit and can involve loss and duplication of messages.

In summary, our networks are generated by the grammar below.

$$\mathsf{N} ::= \overline{x} \langle \tilde{y} \rangle \mid [\mathsf{P}]_A \mid \mathsf{N}_1 | \mathsf{N}_2 \mid (\nu x) \mathsf{N} \mid \mathsf{0}$$

N is well-formed, written $\vdash N$, if $\vdash N$ is derivable using the following rules. (1) $\vdash 0$ is always derivable; (2) $\vdash [P]_A$ if P is local and each free input subject in P is in A; (3) $\vdash N_1 | N_2$ if $\vdash N_1$ and $\vdash N_2$ and, moreover, $ap(N_1) \cap ap(N_2) = \emptyset$; (4) $\vdash (\nu x)N$ if $\vdash N$. Here the access points ap(N) of a network N are given by: $ap([P]_A) = A$, $ap(N_1 | N_2) = ap(N_1) \cup ap(N_2)$ and $ap((\nu x)N) = ap(N) \setminus \{x\}$. The free names of networks are given by $fn(\overline{x}\langle \tilde{y} \rangle) = \{x, \tilde{y}\}$, $fn([P]_A) = fn(P) \cup A$, $fn(M|N) = fn(M) \cup fn(N)$, $fn((\nu x)N) = fn(N) \setminus \{x\}$, $fn(0) = \emptyset$. Bound names are omitted. In the remainder of this text, we assume that expressions involving networks such as $[P]_A$ are well-formed. In particular, quantifications like: "for all P and all A, $[P]_A$ has property X" or even "for all P, $[P]_A$ has property X" abbreviate the statement: "for all P and all A such that $[P]_A$ is well-formed, $[P]_A$ has property X". On networks, \equiv is generated by the axioms below.

$$\begin{split} \mathsf{M} &\equiv_{\alpha} \mathsf{N} \Rightarrow \mathsf{M} \equiv \mathsf{N} \\ \mathsf{L} \mid (\mathsf{M} \mid \mathsf{N}) \equiv (\mathsf{L} \mid \mathsf{M}) \mid \mathsf{N} \\ x \notin \mathsf{fn}(\mathsf{M}) \Rightarrow \mathsf{M} \mid (\nu x) \mathsf{N} \equiv (\nu x)(\mathsf{M} \mid \mathsf{N}) \\ (\nu x) 0 \equiv 0 \\ [0]_{\emptyset} \equiv 0 \end{split} \qquad \qquad \begin{aligned} \mathsf{M} \mid \mathsf{N} \equiv \mathsf{N} \mid \mathsf{M} \\ \mathsf{M} \mid 0 \equiv \mathsf{M} \\ (\nu x)(\nu y) \mathsf{M} \equiv (\nu y)(\nu x) \mathsf{M} \\ (\nu x)(\nu y) \mathsf{M} \equiv (\nu x)[\mathsf{M}]_{\mathcal{A} \cup \{x\}} \\ \mathsf{P} \equiv \mathsf{Q} \Rightarrow [\mathsf{P}]_{\mathcal{A}} \equiv [\mathsf{Q}]_{\mathcal{A}} \end{aligned}$$

One of the key objectives in the design of π_{mlt} was to retain the semantics of the underlying π_t , to allow separation of reasoning about networks from reasoning about processes. Hence the first reduction rule.

 $(INTRA) \quad \mathsf{P} \to \mathsf{Q} \Rightarrow [\mathsf{P}]_A \to [\mathsf{Q}]_A$

Inter-site communication happens by message migration.

$$\begin{array}{ll} (\text{OUT}) & x \notin A, [\overline{x} \langle \tilde{y} \rangle | \mathsf{P}]_A \to [\phi(\mathsf{P})]_A | \overline{x} \langle \tilde{y} \rangle \\ (\text{In}) & x \in A, [\mathsf{P}]_A | \overline{x} \langle \tilde{y} \rangle \to [\mathsf{P} \mid \overline{x} \langle \tilde{y} \rangle]_A \end{array}$$

Incursion of one time step in (OUT) is crucial for a smooth integration of π_t^{loc} into π_{mlt} . Message failures arise from the following rules (which deal with messages in transit only).

$$(\text{Loss}) \quad \overline{x}\langle \tilde{y} \rangle \to 0 \qquad (\text{Dupl}) \quad \overline{x}\langle \tilde{y} \rangle \to \overline{x}\langle \tilde{y} \rangle \,|\, \overline{x}\langle \tilde{y} \rangle$$

Many distributed systems offer only weak guarantees on the upper bound of inter-location clock drift. (PAR) reflects this by not synchronising different sites through application of time-stepping.

$(PAR) \quad M \to M' \Rightarrow M|N \to M'|N$

TEAM LING

The remaining rules are:

(CONG)
$$M \cong M' \to N' \equiv N \Rightarrow M \to N$$
 (Res) $M \to N \Rightarrow (\nu x)M \to (\nu x)N$.

A binary relation \mathcal{R} on processes is a π_{mlt} -congruence if it is an equivalence, if $\equiv \subseteq \mathcal{R}$ and if \mathcal{PRQ} implies $C[\mathcal{P}]\mathcal{R}C[\mathcal{Q}]$ for all network contexts $C[\cdot]$. Network contexts are given by the grammar $C[\cdot] ::= [\cdot] \mid C[\cdot] \mid N \mid (\nu x)C[\cdot]$. Barbs are generated by the following rules. $M \downarrow_x$ and $x \notin ap(N)$ imply $M|N \downarrow_x$, $M \downarrow_x$ and $x \neq a$ imply $(\nu a)M \downarrow_x$ and $\overline{x}\langle \overline{y} \rangle \downarrow_x$. A symmetric binary relation \mathcal{R} on networks is a strong barbed bisimulation if it is a π_{mlt} -congruence and if $M \mathcal{R} N$ implies: (1) for all names $x: M \downarrow_x \Rightarrow N \downarrow_x$; and (2) whenever $M \to M'$ then there is a network N' such that $N \to N'$ and $M' \mathcal{R} N'$. The largest barbed bisimulation $\stackrel{rc}{\sim}$ is called strong reduction congruence. Barbed bisimulation and reduction congruence $\stackrel{rc}{\approx}$ are derived as usual.

Examples (3)

- 1. Let $\mathbf{fw}_{xy} = |x(v).\overline{y}\langle v \rangle$. Then the network $[\overline{x}\langle a \rangle]_{\emptyset} | [\mathbf{fw}_{xy}]_x | [\mathbf{fw}_{yz}]_y | [z(v).Q]_z$ tries to relay the message $\overline{x}\langle a \rangle$ via two intermediate hops to $[z(v).Q]_z$, where it will be used by Q. It can be seen as a distributed version of $\overline{x}\langle a \rangle | \mathbf{fw}_{xy} | \mathbf{fw}_{yz} | z(v).Q$, but semantically it is rather different, due to message loss and duplication.
- 2. The next example shows how to deal with message failure.

 $[(\nu ab)(\overline{x}\langle \tilde{y}a\rangle | \mathsf{timer}^t(a,\overline{b}) | !b.(\overline{x}\langle \tilde{y}a\rangle | \mathsf{timer}^t(a,\overline{b})))|\mathsf{P}]_A | [x(\tilde{\nu}a).(\overline{a} | \mathsf{Q})]_B.$

The location on the left sends a message to that on right and sets a timer to wait for an acknowledgement. If that doesn't come in time, it resends the original message.

3. We can also locate the time services of Example 1(3) as $[delay^n(\mathsf{P}^m)]_A$ but because there is no synchronisation of time between sites, this is not very effective: the location is bisimilar to $[\mathsf{P} \oplus \mathsf{O}]_A$.

The last example is indicative of π_{mlt} 's being too asynchronous for realistic models of distributed systems. In other aspects, too, this calculus is overly idealising, for example in its lack of location failure. The point of π_{mlt} is rather to facilitate the study of message failure in isolation, as a first step towards more realistic models.

3.2 The Maximal Sound Theory

The development in this section mirrors that for π_t with proof being similar, albeit more involved because of possible message failure. π_{mlt} -logics (\mathcal{T}, \vdash) are like π_t -logics, except that formulae are now pairs (M, N) of networks such that ap(M) = ap(N).

As in π_t , there is no maximal consistent and reduction-closed theory. A network M is *insensitive* if $an(N) = \emptyset$ for all reduction sequences $M \rightarrow N$,

where *active names* for networks extend those of processes: $\operatorname{an}(\overline{x}\langle \tilde{y} \rangle) = \{x\}$, $\operatorname{an}([P]_A) = \operatorname{an}(P)$, $\operatorname{an}(M|N) = \operatorname{an}(M) \cup \operatorname{an}(N)$, $\operatorname{an}((\nu x)N) = \operatorname{an}(N) \setminus \{x\}$, $\operatorname{an}(0) = \emptyset$. A theory is *sound* if it is consistent, reduction-closed and identifies all insensitive terms. As before, we set $\mathcal{T}_{\max} = \bigcup \{\mathcal{T} \mid \mathcal{T} \text{ is a sound theory}\}$. \mathcal{T}_{\max} is called the *maximum sound theory*.

Theorem 4. (1) T_{max} is the unique sound theory such that $|\mathcal{T}| \subseteq |T_{max}|$ for all sound theories \mathcal{T} . (2) There is no largest, consistent, reduction-closed theory. (3) $|T_{max}| = T_{max} = \stackrel{rc}{\approx}, \stackrel{rc}{\sim} \subsetneq \stackrel{rc}{\approx}$.

3.3 Labelled Semantics

As with π_t , we present an asynchronous transition system \xrightarrow{l}_{a} . The induced asynchronous bisimilarity soundly approximates T_{max} , but does not characterise it. Characterisation fails because the timers in different sites are not synchronised. The most interesting rule is that for parallel composition

$$\mathsf{M} \xrightarrow{l}_{a} \mathsf{M}', \mathsf{bn}(l) \cap \mathsf{fn}(\mathsf{N}) = \emptyset, (l = \overline{x} \langle (\nu \tilde{y}) \tilde{z} \rangle \Rightarrow x \notin \mathsf{ap}(\mathsf{N})) \Rightarrow \mathsf{M}|\mathsf{N} \xrightarrow{l}_{a} \mathsf{M}'|\mathsf{N}$$

The reason for the side condition $l = \overline{x} \langle (\nu \tilde{y}) \tilde{z} \rangle \Rightarrow x \notin ap(N)$ is that well-formed observers cannot input on channels that are in ap(N).

The remaining rules follow.

$$\begin{split} \mathsf{M} & \stackrel{l}{\longrightarrow}_{a} \mathsf{N}, x \notin \mathsf{fn}(l) \cup \mathsf{bn}(l) \Rightarrow (\nu x) \mathsf{M} \stackrel{l}{\longrightarrow}_{a} (\nu x) \mathsf{N} & 0 \xrightarrow{\mathfrak{X}(\tilde{y})}_{a} \overline{x}\langle \tilde{y} \rangle \\ x \notin A \Rightarrow [\mathsf{P}] \overline{x} \langle \tilde{y} \rangle]_{A} \xrightarrow{\tau}_{a} [\phi(\mathsf{P})]_{A} \mid \overline{x} \langle \tilde{y} \rangle & \overline{x} \langle \tilde{y} \rangle \xrightarrow{\overline{x}\langle \tilde{y} \rangle}_{a} 0 \\ x \in A \Rightarrow [\mathsf{P}]_{A} \mid \overline{x} \langle \tilde{y} \rangle \xrightarrow{\tau}_{a} [\mathsf{P}] \overline{x} \langle \tilde{y} \rangle]_{A} & \overline{x} \langle \tilde{z} \rangle \xrightarrow{\tau}_{a} 0 \\ \mathsf{M} \equiv \mathsf{M}' \xrightarrow{l}_{a} \mathsf{N}' \equiv \mathsf{N} \Rightarrow \mathsf{M} \xrightarrow{l}_{a} \mathsf{N} & \overline{x} \langle \tilde{z} \rangle \xrightarrow{\tau}_{a} \overline{x} \langle \tilde{z} \rangle \mid \overline{x} \langle \tilde{z} \rangle \\ \mathsf{P} \xrightarrow{\tau}_{a} Q \Rightarrow [\mathsf{P}]_{A} \xrightarrow{\tau}_{a} [\mathsf{Q}]_{A} \\ \mathsf{M} \xrightarrow{\overline{x}(\langle \nu \tilde{y} \rangle \overline{z})}_{a} \mathsf{N}, a \neq x, \in \{\tilde{z}\} \setminus \{\tilde{y}\} \Rightarrow (\nu a) \mathsf{M} \xrightarrow{\overline{x}(\langle \nu \tilde{y}, a \rangle \overline{z})}_{a} \mathsf{N} \end{split}$$

A symmetric relation \mathcal{R} is an *strong asynchronous bisimulation* if $(\mathsf{M}, \mathsf{N}) \in \mathcal{R}$ implies whenever $\mathsf{M} \xrightarrow{l}_{a} \mathsf{M}'$ then there is a transition sequence $\mathsf{N} \xrightarrow{l}_{a} \mathsf{N}$ such that $\mathsf{M}' \mathcal{R} \mathsf{N}'$. The largest strong asynchronous bisimulation \sim_a is called strong asynchronous bisimilarity. The largest asynchronous bisimulation \approx_a is defined analogously.

Theorem 5. (1) If $T'_{max} \vdash \mathsf{P} = \mathsf{Q}$ then $[\mathsf{P}]_A \sim_a [\mathsf{Q}]_A$, where T'_{max} is the maximal sound theory on π_t^{loc} ; (2) \approx_a is a congruence; (3) \approx_a is not closed under renaming. (4) $\sim_a \subsetneq \approx_a \subsetneq \approx_a$.

To see that \mathcal{T}_{max} properly includes \approx_a consider $[\mathsf{P}|\mathsf{eq}_{yz}|\overline{x}\langle y\rangle]_A \not\approx_a [\mathsf{P}|\mathsf{eq}_{yz}|\overline{x}\langle z\rangle]_A$ where A contains y, z and P is an arbitrary process. To verify that these two networks are related by \mathcal{T}_{max} , define \mathcal{T} by

$$\begin{split} &[\mathsf{P}|\mathsf{eq}_{yz}|\overline{x}\langle y\rangle|\Pi_{i\in I}\overline{c_{i}}\langle\tilde{d_{i}}\rangle]_{A}|\Pi_{j\in J}\overline{a_{j}}\langle\tilde{b_{j}}\rangle \ \mathcal{T} \ [\mathsf{Q}|\mathsf{eq}_{yz}|\overline{x}\langle y\rangle|\Pi_{i\in I}\overline{c_{i}}\langle\tilde{d_{i}}\rangle]_{A}|\Pi_{j\in J}\overline{a_{j}}\langle\tilde{b_{j}}\rangle \\ &[\mathsf{P}]|\mathsf{eq}_{yz}|\Pi_{i\in I}\overline{a_{i}}\langle\tilde{b_{i}}\rangle]_{A}|\Pi_{j\in J}\overline{c_{j}}\langle\tilde{d_{j}}\rangle \ \mathcal{T} \ [\mathsf{Q}|\mathsf{eq}_{yz}|\Pi_{i\in I}\overline{a_{i}}\langle\tilde{b_{i}}\rangle]_{A}|\Pi_{j\in J}\overline{c_{j}}\langle\tilde{d_{j}}\rangle \end{split}$$

TEAM LING

It is possible but laborious to verify that $\mathcal{T} \cup \{(M,N)|M,N \text{ insensitive}\}$ is a sound theory.

This theorem shows that π_{mlt} integrates and extends π_t^{loc} in a strong sense. Congruency and failure of renaming-closure can coexist because π_{mlt} does not have prefixing operators.

3.4 Locating Processes

How expressive is π_{mlt} compared with π_t ? It might be possible to modify the separation result in [10] to show that π_t cannot (nicely) encode π_{mlt} . The other way round may be more interesting: how is (discretely timed) name-passing affected by message failure? Would it be possible to design a non-distributed process first – without having to worry about distribution – and then scaffold it so that it can function in a distributed setting? This roughly boils down to finding a transformation $(\cdot)^{\bullet}$ that allows to go from non-located, failure-free processes P | Q to $[P^{\bullet}]_A | [Q^{\bullet}]_B$ in a semantics preserving way. Without message failure, that would not be a problem, but loosing messages might lead to deadlocks and duplicated messages may confuse a receiver. We suspect that no appropriate encoding (·)[•] could work for all π_t processes. But that does not mean translations must fail for all processes. As an example of a class of processes that allows distribution, let P, Q be timer free and $x \notin fn(P) \cup fn(Q)$. Assume we wanted to distribute $\mathsf{P} | \overline{x} \langle \widetilde{y} \rangle$ and $x(\widetilde{v}) \cdot \mathsf{Q}$ as $[(\mathsf{P} | \overline{x} \langle \widetilde{y} \rangle)^{\bullet}]_A | [(x(\widetilde{v}) \cdot \mathsf{Q})^{\bullet}]_B$. By the conditions on free names, message duplication is no problem. To overcome message loss, we replace $\overline{x}\langle \tilde{y} \rangle$ with $(\nu ab)(\overline{x}\langle \tilde{y}a \rangle | timer^t(a, \overline{b}) | !b.(\overline{x}\langle \tilde{y}a \rangle | timer^t(a, \overline{b})))$ and $x(\tilde{v})$.Q with $x(\tilde{v}a)$.($\bar{a} \mid Q$) (a, b fresh and ignoring the scaffolding of P and Q for brevity), i.e. we do what TCP does to deal with message loss and add an explicit acknowledgement. If that isn't returned in time, the original message is resent. The resulting distributed process is

$$[\mathsf{P}^{\bullet} | (\nu ab)(\overline{x}\langle \tilde{y}a \rangle | \mathsf{timer}^{t}(a, \overline{b}) | !b.(\overline{x}\langle \tilde{y}a \rangle | \mathsf{timer}^{t}(a, \overline{b})))]_{A} | [x(\tilde{v}a).(\overline{a} | \mathsf{Q}^{\bullet})]_{B}.$$

It is equated by \mathcal{T}_{max} with $[\mathbb{P}^{\bullet}]_A | [\mathbb{Q}^{\bullet}\{\tilde{y}/\tilde{v}\}]_B$ as we sketch later. This translation is quite inefficient, it even introduces divergence, but that does not matter because – due to the absence of inter-site clock synchronisation – \mathcal{T}_{max} is divergence insensitive. More sophisticated variants of our translations are possible, the pragmatically most important being putting an upper bound on the number of retransmissions and making time-out times contingent on the number of failed retransmissions. It would also be possible to dispense with acknowledgement and time-outs altogether: simply use $(\nu a)(\bar{a} | !a.(\bar{x}\langle \tilde{y} \rangle | \bar{a}))$ to flood the receiver with an unbounded number of messages. This brute force approach is semantically sound under the aforementioned constraints, but it has less potential for generalisation and refinement, whether by using less asynchronous equivalences or by limiting the number of retransmissions.

Continuing with the process above, we show that $[\mathbb{P}^{\bullet}]_{A} \mid [\mathbb{Q}^{\bullet}\{\tilde{y}/\tilde{v}\}]_{B}$ is related by \approx_{a} to $[\mathbb{P}^{\bullet}|(\nu ab)(\overline{x}\langle \tilde{y}\rangle|\mathsf{timer}^{t}(a, \overline{b})]!b.(\overline{x}\langle \tilde{y}\rangle|\mathsf{timer}^{t}(a, \overline{b}))]_{A}|[x(\tilde{v}a).(\overline{a}|\mathbb{Q}^{\bullet})]_{B}$. Set $U^{t} = (\mathsf{timer}^{t}(a, \overline{b})|!b.\overline{x}\langle \tilde{y}\rangle|\mathsf{timer}^{t}(a, \overline{b}))$. In addition, let $\mathbb{R} \oplus S$, the *internal sum* of R and S, be the process $(\nu a)(a.R | a.S | \overline{a})$, where a is fresh. Then we can reason in little steps as follows.

$$\begin{split} \left[\mathsf{P}^{\bullet} \mid (\nu ab)(\overline{x}\langle \tilde{y}a \rangle \mid \mathsf{U}^{t}) \right]_{A} \mid [x(\tilde{v}a).(\overline{a} \mid \mathsf{Q}^{\bullet})]_{B} \\ &\equiv (\nu ab)(\left[\mathsf{P}^{\bullet} \mid \overline{x}\langle \tilde{y}a \rangle \mid \mathsf{U}^{t} \right]_{A\cup \{ab\}} \mid [x(\tilde{v}a).(\overline{a} \mid \mathsf{Q}^{\bullet})]_{B}) \\ &\approx_{a} (\nu ab)(\left[\mathsf{P}^{\bullet} \mid \mathsf{U}^{t} \right]_{A\cup \{ab\}} \mid [\overline{x}\langle \tilde{y}a \rangle \mid x(\tilde{v}a).(\overline{a} \mid \mathsf{Q}^{\bullet})]_{B}) \\ &\approx_{a} (\nu ab)(\left[\mathsf{P}^{\bullet} \mid \mathsf{U}^{t} \right]_{A\cup \{ab\}} \mid [\overline{a} \mid \mathsf{Q}^{\bullet} \{ \tilde{y}/ \tilde{v} \}]_{B}) \\ &\approx_{a} (\nu ab)(\left[\mathsf{P}^{\bullet} \mid \mathsf{U}^{t} \mid \overline{a} \right]_{A\cup \{ab\}} \mid [\mathsf{Q}^{\bullet} \{ \tilde{y}/ \tilde{v} \}]_{B}) \\ &= (\nu ab)(\left[\mathsf{P}^{\bullet} \mid !b.(\overline{x}\langle \tilde{y}a \rangle \mid \mathsf{timer}^{t}(a, \overline{b})) \mid \mathsf{timer}^{t}(a, \overline{b}) \mid \overline{a} \right]_{A\cup \{ab\}} \mid [\mathsf{Q}^{\bullet} \{ \tilde{y}/ \tilde{v} \}]_{B}) \\ &\approx_{a} (\nu ab)(\left[\mathsf{P}^{\bullet} \mid !b.(\overline{x}\langle \tilde{y}a \rangle \mid \mathsf{timer}^{t}(a, \overline{b})) \mid 0 \oplus \overline{b} \right]_{A\cup \{ab\}} \mid [\mathsf{Q}^{\bullet} \{ \tilde{y}/ \tilde{v} \}]_{B}) \\ &\approx_{a} (\nu ab)(\left[\mathsf{P}^{\bullet} \mid !b.\mathsf{timer}^{t}(a, \overline{b}) \mid 0 \oplus \overline{b} \right]_{A\cup \{ab\}} \mid [\mathsf{Q}^{\bullet} \{ \tilde{y}/ \tilde{v} \}]_{B}) \\ &\equiv (\nu ab)(\left[\mathsf{P}^{\bullet} \right]_{A\cup \{ab\}} \mid [\mathsf{Q}^{\bullet} \{ \tilde{y}/ \tilde{v} \}]_{B}) \\ &\equiv (\nu ab)(\left[\mathsf{P}^{\bullet} \right]_{A\cup \{ab\}} \mid [\mathsf{Q}^{\bullet} \{ \tilde{y}/ \tilde{v} \}]_{B}) \\ &\approx_{a} [\mathsf{P}^{\bullet}]_{A} \mid [\mathsf{Q}^{\bullet} \{ \tilde{y}/ \tilde{v} \}]_{B} \end{split}$$

The justification of all the individual steps by defining appropriate bisimulations is straightforward, but rather tedious - [4] has all the details.

4 Conclusion

Models of timed computation are legion, we mention [8, 14] in lieu of a comprehensive overview. A close look at the omitted proofs reveals that bound name passing plays no significant role - scope mobility seems orthogonal to timing, at least in this early stage of integration. This promises easy transfer of the presented technology to other timed calculi. Formalisms for distributed computing are also too numerous to survey here. Most closely related are Dpi [24], Nomadic Pict [25] and the Join Calculus [13]. Other influential distributed extensions of π -calculi can be found in [2, 3, 22]. Possibly the most important criticism of π_t is that it is too synchronous, but also too asynchronous for realistic models. Too synchronous because the absence of clock-drift forces many (in)equalities that might be inappropriate, the coincidence of $\stackrel{rc}{\approx}$ and $\stackrel{rc}{\sim}$ being an example. Always allowing time to pass by (IDLE) means that important progress assumptions can be expressed only indirectly, leading to the charge of too much asynchrony. By modifying the time-stepper ϕ , it is possible to express clock-drift, thus coarsening equivalences. Having all timers to be of the form timer^{t n}($x(\tilde{v})$, P, Q) for some fixed n > 1 may also be an important step towards more liberal equalities. Arbitrary progress assumptions can be studied by semantically restricting the set of valid traces. On the network level, π_{mlt} is also too asynchronous because it puts no constraints on inter-site clock-drift. With modern clock-synchronisation algorithms [19] it is possible to push clock-drift under the average inter-site communication latency (which is still many orders of magnitude above the duration of atomic computational steps). By modifying (PAR) at the network level to also apply $\phi(\cdot)$, suitably augmented to allow intersite clock-drift, π_{mlt} may also become more realistic. A multidimensional open problem looming large is the expressive power of π_t and π_{mlt} . One of its most interesting facets is the question if the translation in §3.4 could be refined to allow a larger class of π_t -processes to be mechanically distributed into π_{mlt} .

References

- 1. ABDULLA, P. A., AND JONSSON, B. Verifying programs with unreliable channels. *Info. & Comp. 127*, 2 (1996), 91–101.
- 2. AMADIO, R. M. An asynchronous model of locality, failure, and process mobility. In *Proc. COORDINATION 97* (1997), vol. 1282 of *LNCS*.
- 3. AMADIO, R. M., AND PRASAD, S. Localities and failures. In *Proc. FSTTCS'94* (1994), vol. 880 of *LNCS*.
- 4. BERGER, M. *Towards Abstractions for Distributed Systems*. PhD thesis, Imperial College, London, 2002.
- 5. BERGER, M., AND HONDA, K. The Two-Phase Commit Protocol in an Extended π -Calculus. In *Proc. EXPRESS'00* (2000), vol. 39 of *ENTCS*.
- 6. BERGER, M., HONDA, K., AND YOSHIDA, N. Sequentiality and the π -calculus. In *Proc. TLCA'01* (2001), vol. 2044 of *LNCS*.
- 7. BERGER, M., HONDA, K., AND YOSHIDA, N. Genericity and the π -Calculus. In *Proc. FOSSACS'03* (April 2003), no. 2620 in LNCS, Springer, pp. 103–119.
- 8. BERGSTRA, J. A., PONSE, A., AND SMOLKA, S. A., Eds. Handbook of Process Algebra. Elsevier, 2001.
- 9. BÖRGER, E., AND STÄRK, R. Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, 2003.
- 10. CARBONE, M., AND MAFFEIS, S. On the expressive power of polyadic synchronisation in pi-calculus. In *Proc. EXPRESS'02* (2002), vol. 68 of *ENTCS*.
- 11. CARDELLI, L., AND GORDON, A. Mobile ambients. TCS 240 (2000).
- 12. FOURNET, C., AND GONTHIER, G. A hierarchy of equivalences for asynchronous calculi. In *Proc. ICALP'98* (1998), no. 1443 in LNCS.
- 13. FOURNET, C., GONTHIER, G., LÉVY, J.-J., MARANGET, L., AND RÉMY, D. A Calculus of Mobile Agents. In *Proc. CONCUR* (1996), vol. 1119 of *LNCS*.
- 14. HENNESSY, M. Timed process algebras: a tutorial. Tech. Rep. CS 1993:02, University of Sussex, Computer Science Department, 1993.
- HONDA, K. Two bisimilarities in ν-calculus. Tech. Rep. 92-002, Keio University, Department of Computer Science, 1992.
- 16. HONDA, K., AND TOKORO, M. On asynchronous communication semantics. In *Object-Based Concurrent Computing* (1992), no. 612, in LNCS.
- 17. HONDA, K., AND YOSHIDA, N. On reduction-based process semantics. *TCS 151* (1995).
- 18. HONDA, K., AND YOSHIDA, N. A uniform type structure for secure information flow. In *POPL'02* (2002), ACM Press, pp. 81–92.
- 19. MILLS, D. Time synchronization server. URL http://www.eecis.udel.edu/~ntp/.
- 20. MILNER, R., PARROW, J., AND WALKER, D. A calculus of mobile processes, parts I and II. *Info. & Comp. 100*, 1 (1992).
- 21. MILNER, R., AND SANGIORGI, D. Barbed bisimulation. In *Proc. ICALP'92* (1992), vol. 623 of *LNCS*.
- 22. RIELY, J., AND HENNESSY, M. Distributed processes and location failures. *TCS* 226 (2001).
- 23. SANGIORGI, D., AND WALKER, D. *The* π -*Calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.
- 24. SEWELL, P. Global/local subtyping and capability inference for a distributed picalculus. In *Proc. ICALP* '98 (1998), vol. 1442 of *LNCS*.
- 25. WOJCIECHOWSKI, P. Nomadic Pict: Language and Infrastructure Design for Mobile Computation. PhD thesis, University of Cambridge, 2000.
- 26. YOSHIDA, N., BERGER, M., AND HONDA, K. Strong Normalisation in the π -Calculus. In *Proc. LICS'01* (2001), IEEE, pp. 311–322. The full version to appear in *Journal of Information and Computation*.

Characterizing EF and EX Tree Logics

Mikołaj Bojańczyk¹* and Igor Walukiewicz²

¹ Uniwersytet Warszawski, Banacha 2, 02-097 Warszawa, Poland ² LaBRI, Université Bordeaux I, 351 cours de la Libération, 33405 Talence Cedex, France

Abstract. We characterize the expressive power of EX, EF and EX+EF logics. These are the fragments of CTL built using the respective operators. We give a forbidden pattern characterization of the tree languages definable in these logics. The characterizations give optimal algorithms for deciding if a given tree language is expressible in one of the three logics.

1 Introduction

We consider the definability problem for logics over binary trees: given a tree language decide if it can be expressed by a formula of the logic in question. The main motivation for considering this problem is to understand the expressive power of tree logics. Although a very old question, it has gained new relevance with XML community's burgeoning interest in tree models [8]. Indeed, numerous new formalisms for describing tree properties have been recently proposed.

For words the definability question is well studied and understood. Starting from the celebrated Schutzenberger theorem [12], characterizing star-free word languages by aperiodicity, numerous other language classes have been characterized. In particular, we now have a good understanding of the expressive power of LTL and its fragments [14,18]. This is in sharp contrast with the case of trees where much less is known.

We feel that the major goal in the study of the definability problem for trees is to characterize the expressive power of first-order logic, or equivalently CTL*[1](we consider finite binary trees here). It seems however that this is a difficult problem whose solution demands new tools and expertise. This is why we have decided to consider fragments of CTL* where the problem turns out to be easier. The fragments in question use the operators EX (there is a successor) and EF (there is a descendant). Apart from being a step towards solving the first-order definability problem, these fragments are interesting on their own. The model-checking problem for them is easier than for CTL: for example when a model is given by a BPP [2] or by a a push-down system [16]. The operators EX and EF are also closely related to path operators of XPath [5,4].

^{*} Supported by the European Community Research Training Network GAMES. and Polish KBN grant No. 4 T11C 042 25.

132 M. Bojańczyk and I. Walukiewicz

We prove the definability problem decidable for three logics: EX, EF and EX+EF. These are built by using the eponymous operators along with boolean connectives. Our decision procedures use a sort of forbidden pattern characterizations which are expressed in terms of the minimal leaves-to-root automaton recognizing a given tree language. The resulting algorithms are polynomial in the number of states of the minimal automaton, or to say it differently, in the number of types of the tree language. If, on the other hand, we assume that the input is a CTL formula or a nondeterministic tree automaton then we obtain the EXPTIME upper bound matching the obvious lower bound for the problem.

As mentioned above, not much is known about the definability problem. There exist basic results: characterizations of the class of regular tree languages by monadic second-order logic [15] or the mu-calculus [9]; equivalence of first-order logic and CTL* over finite binary trees [6]. Yet there is no equivalent of the Schutzenberger theorem for trees, indeed the decidability the problem is still open. There has been some work in this direction; in particular borrowing the notion of aperiodicity from the word case is known to be insufficient [11,7]. It is also a valid question to compare the characterizations presented in this paper with the ones in [18] for the corresponding logics for words. Although there is some resemblance between the two, our results need more than a straightforward extension of the forbidden pattern characterizations from the word case. This is in a way unfortunate because it suggests that an equivalent of the Schutzenberger theorem for trees may also require an intricate extension of the aperiodicity.

The plan of the paper is as follows. After a preliminary section we briefly state a characterization of EX logic. This is very similar to a characterization of modal logics presented in the literature [10] so we mention the result mostly for completeness. In the next two sections we characterize the EF and EX+EF logics respectively. Maybe counterintuitively, the argument for the weaker EF logic is longer. In the penultimate section we summarize the results, showing how they imply decidability algorithms. Finally, we justify our characterizations by pointing out why the forbidden patterns known from the word case do not adapt directly to the tree case.

2 Basic Definitions

Let Σ be a finite set called the *alphabet*. We will denote elements of Σ by a, b, c, \ldots and call them *letters*.

A binary tree is a finite prefix-closed subset of $\{0, 1\}^*$ such that for every $v \in \{0, 1\}^*$: v0 is in the tree if and only if v1 is. Allowing vertices with one successor would not change our results but would slightly complicate the notation. The empty sequence ε is the *root* of the tree. For $w, v \in \{0, 1\}^*$, we write w > v if v is a proper prefix of w; we call w a *descendant* of v.

A Σ -tree is a function $t: S \to \Sigma$ where S is a binary tree. By dom(t) we denote the *domain* of t, i.e. S. We use Trees(Σ) to denote the set of Σ -labeled trees. A Σ -language is any subset $L \subseteq \text{Trees}(\Sigma)$.

Given a Σ -tree t and $v \in \text{dom}(t)$, the tree $t|_v : \{w : v \cdot w \in \text{dom}(t)\} \to \Sigma$ is defined by $t|_v(w) = t(v \cdot w)$. For two Σ -trees t_0, t_1 and $a \in \Sigma$, let $a[t_0, t_1]$ denote the unique Σ -tree t such that $t(\varepsilon) = a, t|_0 = t_0$ and $t|_1 = t_1$. The substitution t[v := s] of a tree s in a node v of a tree t is defined in a standard way.

A Σ -multicontext is a tree C over the alphabet $\Sigma \cup \{[]\}$ with the letter [] labeling only leaves, called *holes of* C, For a multicontext C with n holes and Σ -trees t_1, \ldots, t_n the substitution operation $C[t_1, \ldots, t_n]$ is defined in the natural manner (t_1 being substituted in the leftmost hole, etc.) Given a function ν assigning Σ -trees to the holes { v_1, \ldots, v_n } in $C, C[\nu]$ is shorthand for $C[\nu(v_1), \ldots, \nu(v_n)]$. A multicontext with only one hole v is called a *context* and denoted C[].

Two trees s and t are *context equivalent* when for all contexts C[] we have: $C[s] \in L$ if and only if $C[t] \in L$. An *L-type* is an equivalence class of this relation. Types will be denoted by letters: α , β , ... We write $type_L(t)$ for the type of the tree t. Observe that the type of a tree $a[t_0, t_1]$ depends only on the letter a and the types of t_0 and t_1 . This justifies the notation $a[\alpha_0, \alpha_1]$ for some types α_0, α_1 . Similarly we write $t[\beta]$ for the type of the tree t[s] where s is some tree of type β . A language is *regular* if it has a finite number of types.

The set of EX+EF formulas over an alphabet Σ is defined by the following grammar:

$$\mathcal{F} := \varSigma \mid \mathcal{F} \land \mathcal{F} \mid \mathcal{F} \lor \mathcal{F} \mid \neg \mathcal{F} \mid \mathsf{EX}\mathcal{F} \mid \mathsf{EF}\mathcal{F}$$

The operators in the last two productions of this grammar are called the *modalities*. The validity of a formula φ in a tree t, denoted $t \vDash \varphi$, is defined by induction on φ :

 $-t \models a$ if $a = t(\varepsilon)$, for $a \in \Sigma$;

- validity for boolean operations is defined in the standard way;

- $-t \models \mathsf{EX}\varphi$ if there is a node $w \in \{0,1\}$ with $t|_w \models \varphi$;
- $-t \models \mathsf{EF}\varphi$ if there is a node $w > \varepsilon$ with $t|_w \models \varphi$.

Observe that EF has strict semantics. The formula $AX\varphi$ is an abbreviation of $\neg EX \neg \varphi$. The formula $AG\varphi$ is an abbreviation of $\neg EF \neg \varphi$.

Given a set of modalities $\mathcal{M} \subseteq \{\mathsf{EF}, \mathsf{EX}\}\)$, we use $\mathrm{TL}(\mathcal{M})$ for the set of formulas constructed using boolean operations, letter constants and modalities from \mathcal{M} . We say that a language *L* is $\mathrm{TL}(\mathcal{M})$ *definable* if and only if there exists a formula in $\mathrm{TL}(\mathcal{M})$ satisfied in exactly the trees from *L*.

3 TL(EX)

In this section we state a characterization of TL(EX)-definable languages. We do this for the sake of completeness since the characterization is essentially the same as in [10].

Definition 1. Two trees are identical up to depth k if they are the same when restricted to $\{0,1\}^{\leq k}$. We say that a language L is dependent on depth k if every two trees which are identical up to depth k have the same L-type.

A context is *nontrivial* if its hole is not in the root.

Definition 2. Let *L* be a language and let α, β be two distinct *L*-types. We say that the language *L* contains an $\{\alpha, \beta\}$ -loop if for some nontrivial context C[], both $C[\alpha] = \alpha$ and $C[\beta] = \beta$ hold.

Theorem 1. For a regular language L, the follouiing conditions are equivalent:

- 1. L is TL(EX)-definable;
- 2. For some $k \in \mathbb{N}$, L is dependent on depth k;
- 3. L does not have an $\{\alpha, \beta\}$ -loop for any two L-types α, β .

4 TL(EF)

In this section we show a characterization of TL(EF)-definable languages. This is the most involved section of the paper, with a long technical proof.

Before we can formulate the main theorem (Theorem 2) we need some auxiliary definitions. We start with the key definition in this section: that of a delayed type.

Given a Σ -tree t and a letter $a \in \Sigma$, we write $t\langle a \rangle$ to denote the tree obtained from t by relabeling the root with the letter a. With every Σ -tree t we associate its *delayed type*, which is the function:

$$dtype_L(t): \Sigma \to Types(L)$$
 defined $dtype_L(t)(a) = type_L(t\langle a \rangle).$

Note that the delayed type of a tree does not depend on the letter labeling its root. We will denote delayed types using the letters x, y, z. We write $(x, a) \triangleleft_L y$ if there is a tree of delayed type y having a subtree of type x(a). We also write $x \triangleleft_L y$ if $(x, a) \triangleleft_L y$ for some $a \in \Sigma$. This relation is a quasiorder but not necessarily a partial order, since it may not be antisymmetric.

For delayed types x, y and letters $a, b \in \Sigma$, we write $dtype_L(x, a, y, b)$ for the delayed type which assigns to a letter c the type c[x(a), y(b)]. In other words, this is the delayed type of a tree whose left and right subtrees have types x(a) and y(b) respectively. The set of *neutral letters* of a delayed type x is the set

$$N_x^L = \{a : x = dtype_L(x, a, x, a)\}.$$

Definition 3. A Σ -language L is EF-admissible if it is regular and all delayed types x, y and letters $a, c \in \Sigma$ satisfy:

P1 The relation \trianglelefteq_L on delayed types is a partial order; **P2** $dtype_L(x, a, y, b) = dtype_L(x, a, y, b')$ for all $b, b' \in N_y^L$; **P3** if $(x, a) \trianglelefteq_L y$ then $dtype_L(x, a, y, c) = dtype_L(y, c, y, c)$; **P4** $dtype_L(x, a, y, c) = dtype_L(y, c, x, a)$.

Another important concept used in Theorem 2 is that of typeset dependency. The *typeset* of a Σ -tree t is the set

$$\mathrm{TS}_L(t) = \{type_L(t|_w) : w \in \mathrm{dom}(t) \setminus \{\varepsilon\}\}$$
.

TEAM LING

Note that the type of the tree itself is not necessarily included in its typeset. We say that a language L is *typeset dependent* if the delayed type of a tree depends only on its typeset.

Our characterization of TL(EF) is presented in the following theorem:

Theorem 2. For a regular language L, the following conditions are equivalent:

- 1. L is TL(EF)-definable,
- 2. L is typeset dependent,
- 3. L is EF-admissible.

The proof of this theorem is long and will be spread across the next two sections; the implications $1 \Rightarrow 2$ and $3 \Rightarrow 1$ being proved in Sections 4.1 and 4.2 respectively. The implication $2 \Rightarrow 3$ is a simple verification and is omitted. For the remainder of Section 4 we assume that an alphabet Σ along with a Σ -language L are fixed, hence we will drop the L qualifier from the notation, writing for instance \preceq instead of \preceq_L .

4.1 A TL(EF)-Definable Language Is Typeset Dependent

In this section, we will show that the language L is typeset dependent using the assumption that it is defined by some TL(EF) formula ψ .

Definition 4. By $cl(\psi)$ we denote the smallest set of formulas that contains ψ and is closed under negations and subformulas.

It is not difficult to see that the type of a tree is determined by the set of those formulas from $cl(\psi)$ which it satisfies (although this correspondence need not be injective). Our first step is to show that for the delayed type, even less information is sufficient.

Definition 5. An existential formula is a formula of the form $\mathsf{EF}\varphi$. The signature sig(t) of a tree t is the set of existential formulas from $cl(\psi)$ that it satisfies.

Lemma 1. The signature of a tree determines its delayed type.

Proof. Take two trees s and t with the same signatures. For a given letter $a \in \Sigma$, an easy induction on formula size shows that for all $\varphi \in cl(\psi)$:

$$s\langle a \rangle \models \varphi \quad \text{iff} \quad t\langle a \rangle \models \varphi.$$

This is due to the fact that the modality EX is strict. Since the two trees $s\langle a \rangle$ and $t\langle a \rangle$ satisfy the same formulas from $cl(\psi)$, their types must be the same. As the choice of the letter *a* was arbitrary, this implies that the trees *s* and *t* have the same delayed types.

Given two trees t_0, t_1 and a letter $a \in \Sigma$, we write $sig(t_0, t_1)$ instead of $sig(a[t_0, t_1])$. This notation is unambiguous since $sig(a[t_0, t_1])$ does not depend on the letter a.

Given two types α and β , we denote by $dtype(\alpha, \beta)$ the delayed type which assigns to a letter *a* the type $a[\alpha, \beta]$. A type α is *reachable* from a type β , denoted $\beta \preccurlyeq \alpha$, if $C[\beta] = \alpha$ holds for some context C[]. This relation is a quasiorder and we use \approx for the accompanying equivalence relation. The following simple lemma is given without a proof:

Lemma 2. If t' is a subtree of t, then $sig(t',s) \subseteq sig(t,s)$. If $\alpha \preccurlyeq \beta$ then $dtype(\alpha,\beta) = dtype(\beta,\beta)$.

The following lemma shows that for TL(EF)-definable languages, the relation \approx is a congruence with respect to the function $dtype(\alpha, \beta)$:

Lemma 3. If $\alpha_0 \approx \beta_0$ and $\alpha_1 \approx \beta_1$ then $dtype(\alpha_0, \alpha_1) = dtype(\beta_0, \beta_1)$.

Proof. Since a TL(EF)-definable language satisfies $dtype(\alpha, \beta) = dtype(\beta, \alpha)$, it is sufficient to prove the case where $\beta_1 = \alpha_1$. Let C[] be a context such that $C[\alpha_0] = \beta_0$ and let D[] be a context such that $D[\beta_0] = \alpha_0$. All these contexts exist by assumption. Let s_0 be a tree of type α_0 and let s_1 be a tree of type α_1 . Consider the two sequences of trees $\{s_i\}_{i\geq 0}$ and $\{t_i\}_{i\geq 0}$ defined by induction as follows:

$$\begin{array}{ll} s_0 = s_0; \\ t_i = C[s_i] & \text{for } i \geq 0; \\ s_i = D[t_{i-1}] & \text{for } i \geq 1. \end{array}$$

By a simple induction one can prove that for all $i \ge 0$,

$$type(s_i) = \alpha_0$$
 and $type(t_i) = \beta_0$.

By Lemma 2, for all $i \ge 0$

$$sig(s_i, s_1) \subseteq sig(t_i, s_1) \subseteq sig(s_{i+1}, s_1)$$
.

Since there are only finitely many signatures, there must be some i > 0 such that $sig(s_i, s_1) = sig(t_i, s_1)$. Consequently, by Lemma 1, the delayed types $dtype(\alpha_0, \alpha_1)$ and $dtype(\beta_0, \alpha_1)$ are equal.

We are now ready to show that the language L is typeset dependent. Let s and t be two trees with the same typeset. If this typeset is empty, then both trees have one node and, consequently, the same delayed type. Otherwise one can consider the following four types, which describe the sons of s and t:

$$lpha_0 = type(s|_0)$$
 $lpha_1 = type(s|_1)$ $eta_0 = type(t|_0)$ $eta_1 = type(t|_1)$

We need to prove that $dtype(\beta_0, \beta_1) = dtype(\alpha_0, \alpha_1)$. By assumption that the typesets of s and t are equal, both β_0 and β_1 occur in nonroot nodes of s and both α_0 and α_1 occur in nonroot nodes of t. Thus $\beta_0 \preccurlyeq \alpha$ holds for some $\alpha \in \{\alpha_0, \alpha_1\}$ and similarly for β_1, α_0 and α_1 . The result follows from the following case analysis:

- $-\beta_0, \beta_1 \preccurlyeq \alpha$ for some $\alpha \in \{\alpha_0, \alpha_1\}$. By assumption we must have $\alpha \preccurlyeq \beta$ for some $\beta \in \{\beta_0, \beta_1\}$. Hence $\alpha \approx \beta$. By Lemma 3 we get $dtype(\alpha, \alpha) = dtype(\beta, \beta)$. As $\beta_0, \beta_1 \preccurlyeq \alpha \preccurlyeq \beta$, from Lemma 2 we obtain $dtype(\beta_0, \beta_1) = dtype(\beta, \beta)$. Similarly one proves the equality $dtype(\alpha_1, \alpha_2) = dtype(\alpha, \alpha)$.
- $-\alpha_0, \alpha_1 \preccurlyeq \beta$ for some $\beta \in \{\beta_0, \beta_1\}$. As in the case above.
- A short analysis reveals that if neither of the above holds then $\beta_0 \preccurlyeq \alpha_i \preccurlyeq \beta_0$ and $\beta_1 \preccurlyeq \alpha_{1-i} \preccurlyeq \beta_1$ for some $i \in \{0, 1\}$. Therefore $\beta_0 \approx \alpha_i$ and $\beta_1 \approx \alpha_{1-i}$ and an application of Lemma 3 yields the desired result.

4.2 A EF-Admissible Language Is TL(EF)-Definable

We now proceed to the most difficult part of the proof, where a defining TL(EF) formula is found based only on the assumption that the properties P1 to P4 are satisfied. We start by stating a key property of EF-admissible languages which shows the importance of neutral letters.

Lemma 4. If the delayed type of a tree t is y, then its every proper subtree with delayed type y has the root label in N_y .

Proof. Consider some proper subtree $t|_v$ of delayed type y and its root label b = t(v). Let w be the brother of the node v and let z, c be its delayed type and label, respectively. Obviously $(z, c) \leq y$. By property P3 we get dtype(y, b, z, c) = dtype(y, b, y, b) and consequently $dtype(y, b, y, b) \leq y$. As \leq is a partial order by P1 and since $y \leq dtype(y, b, y, b)$ holds by definition, we get dtype(y, b, y, b) = y. Hence b belongs to N_y .

Note that if the trees t and $t|_v$ have delayed type y, then so does the tree $t|_w$ for any w < v, because \leq is a partial order. In particular, the above lemma says that nodes with delayed type y form cones whose non-root elements have labels in N_y .

Formulas Defining Delayed Types. A delayed type x is *definable* if there is some TL(EF) formula θ_x true in exactly the trees of delayed type x.

The construction of the θ_x formulas will proceed by induction on the \triangleleft order. The first step is the following lemma:

Lemma 5. Let y be a delayed type such that all types $z \triangleleft y$ are definable. For every delayed type x there is a TL(EF) formula $fork_y^x$ such that:

$$t \models fork_u^x$$
 iff $dtype(t) = y$ and for all $w > \varepsilon$, $dtype(t|_w) \triangleleft x$.

The proof of this lemma is omitted here. We would only like to point out that some effort is required, since the $fork_y^x$ formula is not allowed to use the EX operator.

We will use this lemma to construct a formula θ_x defining x. For the rest of Section 4.2 we fix the delayed type x and assume that every delayed type $y \triangleleft x$ is definable by a formula θ_y .

The first case is when x has no neutral letters. By Lemma 4, in a tree of delayed type x both sons have delayed types smaller than x, since there are no neutral letters for x. In this case we can set

$$\theta_x = fork_x^x \ . \tag{1}$$

The correctness of this definition follows immediately from Lemma 5.

The definition of θ_x is more involved when the set of neutral letters for x is not empty. The rest of Section 4.2 is devoted to this case.

Consider first the following formula:

$$\theta_{\mathscr{J}} = \left(\mathsf{EF}\bigvee\{b \land \theta_y : y \triangleleft x \land (y, b) \not \trianglelefteq x\}\right) \lor \bigvee\{fork_y^x : y \not \trianglelefteq x\}$$

The intention of this formula is to spell out evident cases when the delayed type of a node cannot be x. The first disjunct says that there is a descendant with a delayed type and a label that prohibit its ancestors to have type x. The second disjunct says that the type of the node is not x but the types of all descendants are $\exists x$. This formula works correctly, however, only when some assumptions about the tree are made. These assumptions use the following definition: a tree t satisfies the property $OK_x(t)$ if

$$dtype(t) \triangleleft x$$
 or $dtype(t) = x$ and $t(\varepsilon) \in N_x$.

Lemma 6. Let t be a tree where $OK_x(t|_v)$ holds for all $v > \varepsilon$. This tree satisfies $\theta_{\mathbf{x}}$ if and only if $dtype(t) \not \leq x$.

Proof. The left to right implication was already discussed and follows from the assumptions on the θ_y formulas used in θ_y and from Lemma 5.

For the right to left implication, let dtype(t) = dtype(y, b, z, c) with y, b, z, c describing delayed types and labels of the nodes 0 and 1 which correspond to the left and right sons of the root. We consider three cases:

- -y = z = x. This is impossible because $OK_x(t|_0)$ and $OK_x(t|_1)$ hold, so the labels a, b must belong to N_x , and thus dtype(t) = x.
- -y = x and $z \triangleleft x$. Since $OK_x(t|_0)$ holds, the label b belongs to N_x . If the inequality $(z, c) \triangleleft x$ were true (which is not necessarily implied by our assumption that $z \triangleleft x$), then by property P3 we would have

$$dtype(t) = dtype(y, b, z, c) = dtype(x, b, z, c) = dtype(x, b, x, b) = x ,$$

a contradiction with $dtype(t) \not \leq x$. Therefore we have $(z, c) \not \leq x$ and hence the first disjunct of θ_{\neq} holds. The case where z = x and $y \triangleleft x$ is symmetric.

 $-y, z \triangleleft x$. In this case the second disjunct in the definition of $θ_{x}$ must hold by Lemma 5.

Let $\theta_{\triangleleft x}$ stand for $\bigvee_{u \triangleleft x} \theta_y$ and consider the formula

$$\varphi_x = \theta_{\triangleleft x} \vee (\neg \theta_{\not a} \land \bigvee \{a : a \in N_x\}) \; .$$

This formula will be used to express the $OK_x(t)$ property. We use AG^* as the non-strict version of AG, i.e. $AG^*\varphi$ is an abbreviation for the formula $\varphi \wedge AG\varphi$.

Lemma 7. A tree t satisfies $AG^*\varphi_x$ iff $OK_x(t|_v)$ holds for all $v \ge \varepsilon$.

Proof. By induction on the depth of the tree t.

⇒ If t satisfies φ_x because it satisfies $\theta_{\triangleleft x}$, then obviously $OK_x(t|_v)$ holds for all $v \ge \varepsilon$. Otherwise we have

$$t(\varepsilon) \in N_x$$
 and $t \nvDash \theta_{\mathscr{A}}$.

By induction assumption, $OK_x(t|_v)$ holds for all $v > \varepsilon$. But then, by Lemma 6, $dtype(t) \leq x$. This, together with $t \nvDash \theta_{\triangleleft x}$ gives dtype(t) = x and hence $OK_x(t)$.

 \Leftarrow Let t be such that $OK_x(t|_v)$ holds for all $v \ge \varepsilon$. By induction assumption, we have $AG\varphi_x$. We need to prove that t satisfies φ_x . If $type(t) \triangleleft x$ holds, then t satisfies $\theta_{\triangleleft x}$ and we are done. Otherwise, as $OK_x(v)$ holds, dtype(t) = xand $t(\varepsilon) \in N_x$. Hence, by Lemma 6, t satisfies the second disjunct in φ_x .

Since the type of a tree can be computed from its delayed type and root label, the following lemma ends the proof that every EF-admissible language is TL(EF) definable:

Lemma 8. Every delayed type is definable.

Proof. By induction on the depth of a delayed type x in the order \leq . If x has no neutral letters then the defining formula θ_x is as in (1). Otherwise, we set the defining formula to be

$$heta_x =
eg heta_{\triangleleft x} \wedge
eg heta_{\not z} \wedge \mathsf{AG} arphi_x \; .$$

Let us show why θ_x has the required properties. By Lemma 7,

$$t \models \mathsf{AG}\varphi_x \quad \text{iff} \quad \operatorname{OK}_x(t|_w) \text{ for all } w > \varepsilon.$$
 (2)

If $t \models \theta_x$ then we get dtype(t) = x using Lemma 6 and (2). For the other direction, if dtype(t) = x then clearly $\neg \theta_{\triangleleft x}$ holds in t. By Lemma 4, $OK_x(t|_w)$ holds for all $w > \varepsilon$, therefore t satisfies $AG\varphi_x$ by (2), and then the formula $\neg \theta_{\not q}$ holds by Lemma 6.

5 TL(EX, EF)

The last logic we consider in this paper is TL(EX, EF). As in the previous sections, we will present a characterization of TL(EX, EF)-definable languages. For the rest of the section we fix an alphabet Σ along with a Σ -language L and will henceforth omit the L qualifier from notation.

Recall the type reachability quasiorder \preccurlyeq along with its accompanying equivalence relation \approx , which were defined on p. 136. The \approx -equivalence class of a type α is called here its *strongly connected component* and is denoted $SCC_L(\alpha)$. We extend the relation \preccurlyeq to SCCs by setting:

$$\Gamma \preccurlyeq \Delta \quad \text{if} \quad \alpha \preccurlyeq \beta \text{ for some } \alpha \in \Gamma \text{ and } \beta \in \Delta;$$

 $\alpha \preccurlyeq \Gamma \quad \text{if} \quad \alpha \preccurlyeq \beta \text{ for some } \beta \in \Gamma.$

We use the standard notational shortcuts, writing $\Gamma \prec \Delta$ when $\Gamma \preccurlyeq \Delta$ but not $\Gamma = \Delta$; similarly for $\alpha \prec \Gamma$.

Let Γ be some SCC and let $k \in \mathbb{N}$. The (Γ, k) -view of a tree t is the tree $view(\Gamma, k, t)$ whose domain is the set of nodes in t at depth at most k and where a node v is labeled by:

- -t(v) if v is at depth smaller than k;
- $type(t|_v)$ if v is at depth k and $type(t|_v) \prec \Gamma$;
- ? otherwise.

Let $views(\Gamma, k)$ denote the set of possible (Γ, k) -views. The intuition behind the (Γ, k) -view of t is that it gives exact information about the tree t for types which are \preccurlyeq smaller than Γ , while for other types it just says "I don't know". The following definition describes languages where this information is sufficient to pinpoint the type within the strongly connected component Γ .

Definition 6. Let $k \in \mathbb{N}$. The language L is (Γ, k) -solvable if every two trees s and t with types in Γ and the same (Γ, k) view have the same type. The language is k-solvable if it is (Γ, k) -solvable for every SCC Γ and it is SCC-solvable if it is k-solvable for some k.

It turns out that SCC-solvability is exactly the property which characterizes the TL(EX, EF)-definable languages:

Theorem 3. A regular language is TL(EX, EF)-definable if and only if it is SCC-solvable.

The proof of this theorem will be presented in the two subsections that follow.

5.1 An SCC-Solvable Language Is TL(EX, EF)-Definable

In this section we show that one can write TL(EX, EF) formulas which compute views. Then, using these formulas and the assumption that *L* is SCC-solvable, the type of a tree can be found.

Fix some k such that L is k-solvable. Let $views(\alpha)$ be the set of possible (Γ, k) -views that can be assumed in a tree of type $\alpha \in \Gamma$. By assumption on L being k-solvable, we have:

Lemma 9. Let t be a tree such that $type(t) \preccurlyeq \alpha$. The type of t is α if and only if its $(SCC(\alpha), k)$ -view belongs to the set $views(\alpha)$.

The following lemma states that views can be computed using TL(EX, EF). We omit the simple proof by induction.

Lemma 10. Suppose that for every type $\beta \prec \Gamma$, there is a TL(EX, EF) formula θ_{β} defining it. Then for every $i \in \mathbb{N}$ and every $s \in views(\Gamma, i)$ there is a formula ψ_s satisfied in exactly the trees whose (Γ, i) -view is s.

We define below a set of views which certainly cannot appear in a tree with a type in a strongly connected component Γ :

$$\begin{split} \text{Bad}(\Gamma) &= \{a[s,t] : s \in \textit{views}(\alpha), t \in \textit{views}(\beta), \text{where } \alpha, \beta \preccurlyeq \Gamma, a[\alpha, \beta] \not\preccurlyeq \Gamma \} \cup \\ \{t : type(t) \not\preccurlyeq \Gamma \text{ and } \text{dom}(t) = \{\varepsilon\} \} \end{split}$$

Observe that $\text{Bad}(\Gamma)$ is a set of $(\Gamma, k+1)$ -views. The following lemma shows that the above cases are essentially the only ones.

Lemma 11. For a tree t and an SCC Γ , the following equivalence holds:

 $type(t) \not\preccurlyeq \Gamma \quad iff \quad view(\Gamma, k+1, t|_v) \in Bad(\Gamma) \text{ for some } v \in dom(t).$

Proof. Both implications follow easily from Fact 9 if one considers the maximal possible node v satisfying the right hand side.

The following lemma completes the proof that L is TL(EX, EF)-definable.

Lemma 12. Every type of L is TL(EX, EF)-definable.

Proof. The proof is by induction on depth of the type in the quasiorder \preccurlyeq . Consider a type α and its SCC Γ . By induction assumption, for all types $\beta \prec \Gamma$, there is a formula θ_{β} which is satisfied in exactly the trees of type β . Using the θ_{β} formulas and Lemma 10 we construct the following TL(EX, EF) formula (recall that AG^{*} is the non-strict version of AG defined on page 138):

$$\theta_{\Gamma} = \mathsf{AG}^* \bigwedge_{t \in \mathrm{Bad}(\Gamma)} \neg \psi_t.$$

By Lemma 11, a tree t satisfies θ_{Γ} if and only if $type(t) \preccurlyeq \Gamma$. Finally, the formula θ_{α} is defined:

$$\theta_{\alpha} = \theta_{\Gamma} \wedge \bigvee_{t \in views(\alpha)} \psi_t.$$

The correctness of this construction follows from Fact 9.

5.2 A TL(EX, EF)-Definable Language Is SCC-Solvable

In this section, we are going to show that a language which is not SCC-solvable is not TL(EX, EF)-definable. For this, we introduce an appropriate Ehrenfeucht-Fraïsé game, called the EX+EF game, which characterizes trees indistinguishable by TL(EX, EF)-formulas.

The game is played over two trees and by two players, Spoiler and Duplicator. The intuition is that in the *k*-round EX+EF game, the player Spoiler tries to differentiate the two trees using *k* moves.

The precise definition is as follows. At the beginning of the *k*-round game, with $k \ge 0$, the players are faced with two trees t_0 and t_1 . If these have different root labels, Spoiler wins. If they have the same root labels and k = 0, Duplicator wins; otherwise the game continues. Spoiler first picks one of the trees t_i , with $i \in \{0, 1\}$. Then he chooses whether to make an EF or EX move. If he chooses to make EF move, he needs to choose some non-root node $v \in \text{dom}(t_i)$ and Duplicator must respond with a non-root node $w \in \text{dom}(t_{1-i})$ of the other tree. If Spoiler chooses to make an EX move, he picks a son $v \in \{0, 1\}$ of the root in

 t_i and Duplicator needs to pick the same son w = v in the other tree. If a player cannot find an appropriate node in the relevant tree, this player immediately looses. Otherwise the trees $t_i|_v$ and $t_{1-i}|_w$ become the new position and the (k-1)-round game is played.

The following lemma is proved using a standard induction:

Lemma 13. Duplicator wins the k-round EX+EF game over t_0 and t_1 iff t_0 and t_1 satisfy the same EX+EF formulas of modality nesting depth k.

For two types $\alpha, \beta \in \Gamma$ we define an (α, β) -context to be a multicontext *C* such that there are two valuations of its holes $\nu_{\alpha}, \nu_{\beta} : V \to \Gamma$ giving the types $C[\nu_{\alpha}] = \alpha$ and $C[\nu_{\beta}] = \beta$. The hole depth of a multicontext *C* is the minimal depth of a hole in *C*. A multicontext *C* is *k*-bad for an SCC Γ if it has hole depth at least *k* and is an (α, β) -context for two different types $\alpha, \beta \in \Gamma$.

Lemma 14. *L* is not SCC-solvable if and only if for some SCC Γ and every $k \in \mathbb{N}$, it contains multicontexts which are *k*-bad for Γ .

Proof. A k-bad context exists for Γ if and only if L is not (Γ, k) -solvable.

The following lemma concludes the proof that no TL(EX, EF) formula can recognize a language which is not SCC-solvable:

Lemma 15. If L is not SCC-solvable then for every k there are trees $s \in L$ and $t \notin L$ such that Duplicator wins the k-round EX+EF game over s and t.

Proof. Take some $k \in \mathbb{N}$. If *L* is not SCC-solvable then, by Lemma 14, there is a multicontext *C* which is *k*-bad for some SCC Γ . Let $V = \{v_1, \ldots, v_n\}$ be the holes of *C*, let $\nu_{\alpha}, \nu_{\beta} : V \to \Gamma$ be the appropriate valuations and $\alpha = C[\nu_{\alpha}], \beta = C[\nu_{\beta}]$ the resulting types. We will use this multicontext to find trees $s \in L$ and $t \notin L$ such that Duplicator wins the *k*-round EX+EF game over *s* and *t*.

Since all the types used in the valuations ν_{α} and ν_{β} come from same SCC, there are contexts $C_1^{\alpha}[], \ldots, C_n^{\alpha}[]$ and $C_1^{\beta}[], \ldots, C_n^{\beta}[]$ such that

$$C_i^{\alpha}[\alpha] = \nu_{\beta}(v_i)$$
 $C_i^{\beta}[\beta] = \nu_{\alpha}(v_i)$ for all $i \in \{1, \dots, n\}$.

This means there are two contexts D^{α} and D^{β} with *n* holes each, such that: 1) D^{α} and D^{β} agree over nodes of depth less than k; 2) when all holes of D^{α} are plugged with β , we get the type α ; and 3) when all holes of D^{β} are plugged with α , we get the type β . These are obtained by plugging the appropriate "translators" $C_i^{\alpha}[]$ and $C_i^{\beta}[]$ into the holes of the multicontext *C*. Let t_0 be some tree of type α . The trees t_j for j > 0 are defined by induction as follows:

$$t_{2i+1} = D^{\beta}[\underbrace{t_{2i}, \dots, t_{2i}}_{n \text{ times}}]$$
 $t_{2i+2} = D^{\alpha}[\underbrace{t_{2i+1}, \dots, t_{2i+1}}_{n \text{ times}}].$

By an obvious induction, all the trees t_{2i} have type α and all the trees t_{2i+1} have type β . As $\beta \neq \alpha$, there exists a context D[] such that $D[\alpha] \in L$ and $D[\beta] \notin L$ (or the other way round).

To finish the proof of the lemma, we will show that Duplicator wins the k-round EX+EF game over the trees

$$s = D[t_{2k+2}]$$
 and $t = D[t_{2k+1}]$.

The winning strategy for Duplicator is obtained by following an invariant. This invariant is a disjunction of three properties, one of which always holds when the *i*-round game is about to be played:

- 1. The two trees are identical;
- 2. The two trees are $s|_v$ and $t|_v$ for some $|v| \le k i$;
- 3. The two trees are $t_m|_v$ and $t_{m-2}|_v$ for

 $m \geq k+i+1$ and $egin{cases} v \in \mathrm{dom}(D^lpha) & ext{if } m ext{ is even;} \ v \in \mathrm{dom}(D^eta) & ext{if } m ext{ is odd.} \end{cases}$

The invariant holds at the beginning of the first round, due to 2, and one can verify that Duplicator can play in such a way that it is satisfied in all rounds. Item 2 of the invariant will be preserved in the initial fragment of the game when only EX moves are made, then item 3 will hold until either the game ends or item 1 begins to hold.

6 Decidability

In this section we round up the results by showing that our characterizations are decidable.

Theorem 4. It is decidable in time polynomial in the number of types if a language is:

- TL(EX)-definable;
- TL(EF)-definable;
- TL(EX,EF)-definable.

Proof. Using a simple dynamic algorithm, one can compute in polynomial time all tuples $(\alpha, \beta, \alpha', \beta')$ such that for some context C[], $C[\alpha] = \alpha'$ and $C[\beta] = \beta'$. Using this, we can find in polynomial time:

- Whether L contains an $\{\alpha, \beta\}$ -loop;
- The \preccurlyeq_L and \approx_L relations on types.

Since the delayed type of a tree depends only on the types of its immediate subtrees, the number of delayed types is polynomial in the number of types. The relation \triangleleft_L on delayed types can then be computed in polynomial time from the relation \preccurlyeq_L . Having the relations \preccurlyeq_L and \triangleleft_L , one can check in polynomial time if *L* is EF-admissible.

This, along with the characterizations from Theorems 1 and 2, proves decidability for TL(EX) and TL(EF). The remaining logic is TL(EX, EF).

By Theorem 3, it is enough to show that SCC-solvability is decidable. In order to do this, we give an algorithm that detects if a given SCC Γ admits bad multicontexts of arbitrary size, cf. Lemma 14. Fix an SCC Γ . We define by induction a sequence B^i of subsets of $\Gamma \times \Gamma$.

 $-B^0 = \Gamma \times \Gamma.$

 $(\alpha,\beta) \in B^{i+1}$ if $(\alpha,\beta) \in B^i$ and either

- there is a pair $(\alpha', \beta') \in B^i$, a type $\gamma \prec \Gamma$ and a letter $a \in \Sigma$ such that $type(a[\alpha', \gamma]) = \alpha$ and $type(a[\beta', \gamma]) = \beta$; or • there are pairs $(\alpha', \beta'), (\alpha'', \beta'') \in B^i$ and a letter $a \in \Sigma$ such that
- $type(a[\alpha', \alpha'']) = \alpha$ and $type(a[\beta', \beta'']) = \beta$

The sequence B^i is decreasing so it reaches a fix-point B^{∞} in no more than $|\Gamma|^2$ steps. The following lemma yields the algorithm for TL(EX, EF) and concludes the proof of Theorem 4:

Lemma 16. Γ admits bad multicontexts of arbitrary size iff $B^{\infty} \neq \emptyset$.

Corollary 1. If the input is a CTL formula or a nondeterministic tree automaton, all of the problems in Theorem 4 are EXPTIME-complete.

Proof. Since, in both cases, the types can be computed in time at most exponential in the input size, the EXPTIME membership follows immediately from Theorem 4. For the lower bound, one can use an argument analogous to the one in [17] and reduce the EXPTIME-hard universality problems for both CTL [3] and nondeterministic automata [13] to any of these problems.

7 **Open Problems**

The question of definability for the logics TL(EX), TL(EF) and TL(EX, EF) has been pretty much closed in this paper. One possible continuation are logics where instead of EF, the non-strict modality EF* is used. The resulting logics are weaker than their strict counterparts (for instance the language EFa is not definable in TL[EF*]) and therefore decidability of the their definability problems can be investigated. Another question is what happens if we enrich these logics with past quantification (there exists a point in the past)? This question is particularly relevant in the case of TL(EX, EF), since the resulting logic coincides with firstorder logic with two variables (where the signature contains < and two *binary* successor relations). Finally, there is the question for CTL. Note that on words CTL collapses to LTL and hence first-order logic, so such a characterization would subsume first-order definability for words.

References

1. E. A. Emerson and J. Y. Halpern. 'Sometimes' and 'not never' revisited: on branching versus linear time temporal logic. Journal of the ACM, 33(1):151-178, 1986.

- 2. J. Esparza. Decidability of model-checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.
- 3. M. Fischer and R. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18:194–211, 1979.
- 4. M. Franceschet, L. Afanasiev, M. de Rijke, and M. Marx. CTL model checking for processing simple XPath queries. In *Temporal Presentation and Reasoning*.
- 5. C. Koch G. Gottlob. Monadic queries over tree-structured data. In *Logic in Computer Science*, pages 189–202, 2002.
- 6. T. Hafer and W. Thomas. Computation tree logic CTL and path quantifiers in the monadic theory of the binary tree. In *International Colloquium on Automata, Languages and Programming*, volume 267 of *Lecture Notes in Computer Science*, pages 260–279, 1987.
- U. Heuter. First-order properties of trees, star-free expressions, and aperiodicity. In Symposium on Theoretical Aspects of Computer Science, volume 294 of Lecture Notes in Computer Science, pages 136–148, 1988.
- F. Neven. Automata, logic, and XML. In Julian C. Bradfield, editor, *Computer Science Logic*, volume 2471 of *Lecture Notes in Computer Science*, pages 2–26, 2002.
- 9. D. Niwiński. Fixed points vs. infinte generation. In Logic in Computer Science, pages 402–409, 1988.
- M. Otto. Eliminating recursion in the μ-calculus. In Symposium on Theoretical Aspects of Computer Science, volume 1563 of Lecture Notes in Computer Science, pages 531–540, 1999.
- A. Potthoff. First-order logic on finite trees. In *Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 125–139, 1995.
- M. P. Schützenberger. On finite monoids having only trivial subgroups. *Informa*tion and Control, 8:190–194, 1965.
- 13. H. Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19:424–437, 1990.
- 14. H. Straubing. *Finite Automata, Formal Languages, and Circuit Complexity.* Birkhäuser, Boston, 1994.
- 15. J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, 1968.
- I. Walukiewicz. Model checking CTL properties of pushdown systems. In Foundations of Software Technology and Theoretical Computer Science, volume 1974 of Lecture Notes in Computer Science, pages 127–138, 2000.
- 17. I. Walukiewicz. Deciding low levels of tree-automata hierarchy. In *Workshop on Logic, Language, Information and Computation,* volume 67 of *Electronic Notes in Theoretical Computer Science,* 2002.
- T. Wilke. Classifying discrete temporal properties. In Symposium on Theoretical Aspects of Computer Science, volume 1563 of Lecture Notes in Computer Science, pages 32–46, 1999.

Message-Passing Automata Are Expressively Equivalent to EMSO Logic

Benedikt Bollig^{1*} and Martin Leucker^{2**}

 ¹ Lehrstuhl für Informatik II, RWTH Aachen, Germany bollig@informatik.rwth-aachen.de
 ² IT department, Uppsala University, Sweden leucker@it.uu.se

Abstract. We study the expressiveness of finite message-passing automata with a priori unbounded FIFO channels and show them to capture exactly the class of MSC languages that are definable in existential monadic second-order logic interpreted over MSCs. Moreover, we prove the monadic quantifier-alternation hierarchy over MSCs to be infinite and conclude that the class of MSC languages accepted by message-passing automata is not closed under complement. Furthermore, we show that satisfiability for (existential) monadic seconder-order logic over MSCs is undecidable.

1 Introduction

A common design practice when developing communicating systems is to start with drawing scenarios showing the intended interaction of the system to be. The standardized notion of *message sequence charts* (MSCs, [7]) is widely used in industry to formalize such typical behaviors.

An MSC depicts a single partially-ordered execution sequence of a system. It defines a set of processes interacting with one another by communication actions. In the visual representation of an MSC, processes are drawn as vertical lines that are interpreted as time axes. A labeled arrow from one line to a second corresponds to the communication events of sending and receiving a message. Collections of MSCs are used to capture the scenarios that a designer might want the system to follow or to avoid. Several specification formalisms have been considered, such as *high-level MSCs* or *MSC graphs* [2,14].

The next step in the design process usually is to derive an implementation of the system to develop [5], preferably automatically. In other words, we are interested in generating a distributed automaton *realizing* the behavior given in

^{*} Part of this work was done while the author was on leave at the School of Computer Science, University of Birmingham, United Kingdom, and supported by the German Academic Exchange Service (DAAD).

^{**} Supported by the European Research Training Network "Games".

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 146-160, 2004.

form of scenarios. This problem asks for the study of automata models that are suited for accepting the system behavior described by MSC specifications.

A common model that reflects the partially-ordered execution behavior of MSCs in a natural manner are *message-passing automata*, MPAs for short. They consist of several components that communicate using channels. Several variants of MPAs have been studied in the literature: automata with a single or multiple initial states, with finitely or infinitely many states, bounded or unbounded channels, and systems with a global or local acceptance condition.

We focus on MPAs with a priori unbounded FIFO channels and global acceptance condition where each component employs a finite state space. Our model subsumes the one studied in [5] where a local acceptance condition is used. It coincides with the one used in [6,9], although these papers characterize the fragment of channel-bounded automata. It extends the setting of [1,12] in so far as we provide synchronization messages and a global acceptance condition to have the possibility to coordinate rather autonomous processes. Thus, our version covers most existing models of communicating automata for MSCs.

A fruitful way to study properties of automata is to establish logical characterizations. For example, finite word automata are known to be expressively equivalent to monadic second-order (MSO) logic over words. More precisely, the set of words satisfying some MSO formula can be defined by a finite automaton and vice versa. Since then, the study of automata models for generalized structures such as graphs or, more specifically, labeled partial orders and their relation to MSO logic has been a research area of great interest aiming at a deeper understanding of their logical and algorithmic properties (see [16] for an overview).

In this paper, we show that MPAs accept exactly those MSC languages that are definable within the existential fragment of MSO (over MSCs), abbreviated by EMSO. We recall that emptiness for MPAs is undecidable and conclude that so is satisfiability for EMSO and universality for MSO logic.

Furthermore, we show that MSO is strictly more expressive than EMSO. More specifically, the monadic quantifier-alternation hierarchy turns out to be infinite. Thus, MPAs do *not* necessarily accept a set of MSCs defined by an MSO formula. Furthermore, we use this result to conclude that the class of MSC languages that corresponds to MPAs is not closed under complementation, answering the question posed in [9].

MPAs with a priori unbounded channels have been rather used as a model to implement a given (high-level) MSC specification [5]. Previous results lack an algebraic or logical characterization of the corresponding class of languages. They deal with MPAs and sets of MSCs that make use only of a bounded part of the actually unbounded channel [6,9]. More specifically, when restricting to sets of so-called *bounded* MSCs, MSO captures exactly the class of those MSC languages that correspond to some bounded MPAs.

Organization of the Paper. The next two sections introduce some basic notions and recall the definition of message sequence charts and (existential) monadic second-order logic. Section 4 deals with message-passing automata and their expressive equivalence to existential monadic second-order logic, while Section 5 studies the gap between monadic second-order formulas and their existential fragment.

Acknowledgment. We would like to thank Dietrich Kuske for valuable remarks and pointing out some innaccuracies in a previous version of this paper. We also thank the anonymous referees for their helpful suggestions and comments.

2 **Message Sequence Charts**

Forthcoming definitions are all made wrt. a fixed finite set \mathcal{P} of at least two processes. (Note that, in one proof, we assume the existence of at least three processes.) We denote by Ch the set $\{(p,q) \mid p,q \in \mathcal{P}, p \neq q\}$ of reliable FIFO channels. Thus, a message exchange is allowed between distinct processes only. Let Act[!] denote the set $\{p|q \mid (p,q) \in Ch\}$ of send actions while Act[?] denotes the set $\{q?p \mid (p,q) \in Ch\}$ of *receive actions*. Hereby, p!q and q?p are to be read as p sends a message to q and q receives a message from p, respectively. They are related in the sense that they will label communicating events of an MSC, which are joint by a message arrow in its graphical representation. Accordingly, let $Com := \{(p!q, q?p) \mid (p,q) \in Ch\}$. Observe that an action $p\theta q \ (\theta \in \{!, ?\})$ is performed by process p, which is indicated by $P(p\theta q) = p$. We let Act stand for the union of $Act^{!}$ and $Act^{?}$ and, for $p \in \mathcal{P}$, set Act_{p} to be the set $\{\sigma \in Act \mid \sigma Act \mid \sigma \in Act \mid \sigma Act \mid \sigma \in Act \mid A$ $P(\sigma)=p\}.$

For a total order \leq on a finite set *E*, \triangleleft denotes the *covering relation* of \leq : for $e, e' \in E, e < e'$ if both e < e' and, for any $e'' \in E, e < e'' \le e'$ implies e'' = e'.

Definition 1 (Message Sequence Chart). A message sequence chart (MSC) is a structure $(E, \{ \leq_p \}_{p \in \mathcal{P}}, <_{c}, \lambda)$ such that

- -E is a nonempty finite set of events,
- $-\lambda: E \rightarrow Act$ is a labeling function,
- $\lessdot_{\mathbf{p}}$ is the covering relation of some total order $\leq_{\mathbf{p}}$ on $E_{\mathbf{p}} := \{e \in E \mid \lambda(e) \in E\}$ Act_{p}
- $<_{\mathbf{c}} \subseteq E \times E$ such that, for any $e, e' \in E$, $e <_{\mathbf{c}} e'$ iff $(\lambda(e), \lambda(e')) \in Com$ and $|\downarrow e \cap \lambda^{-1}(\lambda(e))| = |\downarrow e' \cap \lambda^{-1}(\lambda(e'))|$ (where, for $e \in E$, $\downarrow e$ is the set of events $e' \in E_{P(\lambda(e))}$ with $e' \leq_{P(\lambda(e))} e$, $- (<_{c} \cup \bigcup_{p \in \mathcal{P}} <_{p})^{*}$ is a partial order, and $- |\lambda^{-1}(p!q)| = |\lambda^{-1}(q?p)|$ for each $(p,q) \in Ch$.

Thus, events on one and the same process line are totally ordered, and events on distinct process lines that communicate with each other in a FIFO manner (wrt. $<_{c}$) are labeled with actions related by *Com*.

Given an MSC $(E, \{ \leq_p \}_{p \in \mathcal{P}}, <_c, \lambda)$ and $e \in E, P(e)$ will serve as a shorthand for $P(\lambda(e))$. The set of MSCs is denoted by MSC and a subset of MSC is called an MSC language.

Henceforth, we identify a structure of any kind with its isomorphism class.

3 (Existential) Monadic Second-Order Logic

Given supplies $Var = \{x, y, ..., x_1, x_2, ...\}$ of *individual variables* and $VAR = \{X, Y, ..., X_1, X_2, ...\}$ of *set variables*, formulas from MSO, the set of *monadic second-order formulas* (over MSCs) are built up from the atomic formulas

 $\lambda(x) = \sigma \text{ (for } \sigma \in Act) \qquad x \in X \qquad x \lessdot_p y \text{ (for } p \in \mathcal{P}) \qquad x \prec_c y \qquad x = y$

(where $x, y \in \text{Var}$ and $X \in \text{VAR}$) and furthermore allow the Boolean connectives $\neg, \lor, \land, \rightarrow$ and the quantifiers \exists, \forall , which can be applied to either kind of variable.

Let $M = (E, \{ \leq_p \}_{p \in \mathcal{P}}, <_c, \lambda)$ be an MSC. Given an interpretation function \mathcal{I} , which assigns to an individual variable x an event $\mathcal{I}(x) \in E$ and to a set variable X a set of events $\mathcal{I}(X) \subseteq E$, the satisfaction relation $M \models_{\mathcal{I}} \varphi$ for a formula φ is given by $M \models_{\mathcal{I}} \lambda(x) = \sigma$ if $\lambda(\mathcal{I}(x)) = \sigma$, $M \models_{\mathcal{I}} x <_p y$ if $\mathcal{I}(x) <_p \mathcal{I}(y)$, and $M \models_{\mathcal{I}} x <_c y$ if $\mathcal{I}(x) <_c \mathcal{I}(y)$, while the remaining operators are defined as usual.

For an MSO formula φ , the notation $\varphi(x_1, \ldots, x_m, X_1, \ldots, X_n)$ shall indicate that at most the variables $x_1, \ldots, x_m, X_1, \ldots, X_n$ occur free in φ . An MSO formula is called *existential* if it is of the form $\exists X_1 \ldots \exists X_n \varphi(X_1, \ldots, X_n, \overline{Y})$ where \overline{Y} is a block of second-order variables and $\varphi(X_1, \ldots, X_n, \overline{Y})$ is a first-order formula. Let EMSO denote the class of existential MSO formulas. In general, Σ_k shall contain MSO formulas of the form $\exists \overline{X_1} \forall \overline{X_2} \ldots \exists / \forall \overline{X_k} \varphi(\overline{X_1}, \ldots, \overline{X_k}, \overline{Y})$ with first-order kernel $\varphi(\overline{X_1}, \ldots, \overline{X_k}, \overline{Y})$ (again, $\overline{X_i}$ and \overline{Y} are blocks of secondorder variables)¹.

In the following sections, we usually consider MSO sentences, i.e., formulas without free variables, and accordingly replace $\models_{\mathcal{I}}$ with \models . For an MSO sentence φ , the *MSC language* of φ , denoted by $L(\varphi)$, is the set of MSCs *M* with $M \models \varphi$. For a set of MSO formulas \mathfrak{L} , an MSC language *L* is called \mathfrak{L} -definable if $L = L(\varphi)$ for some sentence $\varphi \in \mathfrak{L}$. We will show in a subsequent section that the classes of Σ_k -definable languages form an infinite hierarchy when formulas are interpreted over MSCs, resuming a result by Matz and Thomas, who proved infinity of the hierarchy for grids [11]. In other words, the more alternation depth secondorder quantification allows, the more expressive formulas become. However, it will turn out that, to cover the feasible area of realizable MSC languages (in terms of message-passing automata), we can restrict to EMSO-definable MSC languages. The class of MSO-definable MSC languages is denoted by \mathcal{MSO} , the one of EMSO-definable languages by \mathcal{EMSO} .

4 Message-Passing Automata and Their Expressiveness

In this section, we study distributed automata, called *message-passing automata*, which, as we will see, generate MSC languages in a natural manner.

¹ Note that Σ_1 and EMSO coincide.

A message-passing automaton is a collection of finite-state machines that shaxe one global initial state and several global final states. The machines are connected pairwise with a priori unbounded reliable FIFO buffers. The transitions of each component are labeled with send or receive actions. A send action p!q puts a message at the end of the channel from p to q. A receive action can be taken provided the requested message is found in the channel. To extend the expressive power, message-passing automata can send certain synchronization messages. Let us be more precise:

Definition 2 (Message-Passing Automaton). A message-passing automaton (MPA) is a structure $\mathcal{A} = ((\overline{\mathcal{A}}_p)_{p \in \mathcal{P}}, \mathcal{D}, \overline{s}^{in}, F)$ such that

- $-\mathcal{D}$ is a nonempty finite set of synchronization messages (or data),
- for each $p \in \hat{\mathcal{P}}$, $\hat{\mathcal{A}}_p$ is a pair (\hat{S}_p, Δ_p) where
 - S_p is a nonempty finite set of (p) local states and
 - $\Delta_p \subseteq S_p \times Act_p \times \mathcal{D} \times S_p$ is the set of (p-)local transitions,
- $-\overline{s}^{in} \in \prod_{p \in \mathcal{P}} S_p$ is the global initial state, and
- $-F \subseteq \prod_{p \in \mathcal{P}} S_p$ is the set of global final states.

For a global state $\overline{s} = (s_p)_{p \in \mathcal{P}} \in \prod_{p \in \mathcal{P}} S_p$ of $\mathcal{A}, \overline{s}[p]$ will henceforth refer to s_p . We now define the behavior of message-passing automata and, in doing so, adhere to the style of [9]. In particular, an automaton will run on MSCs rather than on linearizations of MSCs, allowing for its distributed behavior. Let $\mathcal{A} =$ $((\mathcal{A}_p)_{p\in\mathcal{P}}, \mathcal{D}, \overline{s}^{in}, F), \ \mathcal{A}_p = (S_p, \Delta_p), \text{ be an MPA and } M = (E, \{ \leq_p \}_{p\in\mathcal{P}}, <_c, \lambda)$ be an MSC. For a function $r : E \to \bigcup_{p\in\mathcal{P}} S_p$, we define $r^- : E \to \bigcup_{p\in\mathcal{P}} S_p$ to map an event $e \in E$ onto $\overline{s}^{in}[P(e)]$ if e is minimal wrt. $\leq_{P(e)}$ and, otherwise, onto r(e') where $e' \in E_{P(e)}$ is the unique event with $e' <_{P(e)} e$. A run of \mathcal{A} on Mis a pair (r, m) of mappings $r : E \to \bigcup_{p \in \mathcal{P}} S_p$ with $r(e) \in S_{P(e)}$ for each $e \in E$ and $m : \langle_c \to \mathcal{D}$ such that, for any $e, e' \in E$, $e <_c e'$ implies

 $-(r^{-}(e),\lambda(e),m((e,e')),r(e)) \in \Delta_{P(e)} \text{ and } \\ -(r^{-}(e'),\lambda(e'),m((e,e')),r(e')) \in \Delta_{P(e')}.$

For $p \in \mathcal{P}$, let f_p denote $\overline{s}^{in}[p]$ if E_p is empty. Otherwise, let f_p denote r(e) where $e \in E_p$ is the maximal event wrt. \leq_p . We call (r, m) accepting if $(f_p)_{p\in\mathcal{P}}\in F.$

For an MPA \mathcal{A} , we denote by $L(\mathcal{A}) := \{M \in \mathbb{MSC} \mid \text{there is an accepting run}\}$ of \mathcal{A} on M the language of \mathcal{A} . Let furthermore $\mathcal{MPA} := \{L \subseteq \mathbb{MSC} \mid L = L(\mathcal{A})\}$ for some MPA \mathcal{A} denote the class of languages that are *realizable* as MPAs.

Remark 1. The emptiness problem for MPAs is undecidable.

Proof. Several decidability questions were studied for *communicating finite-state* machines, a slightly different variant of MPAs. Among them, (a problem related to) the emptiness problem for communicating finite-state machines turned out to be undecidable [3]. The proof can be easily adapted towards MPAs.

We now turn towards one of our main results and first mention that an MPA can be effectively transformed into an equivalent EMSO sentence.

Lemma 1. $MPA \subseteq EMSO$

Proof. Several instances of this problem have been considered in the literature and can be easily adapted to our setting. See [17], for example. \Box

Corollary 1. The following two problems are undecidable:

- (a) Satisfiability for EMSO sentences over MSC
- (b) Universality for MSO sentences over MSC

Proof. Using Remark 1 and Lemma 1, we obtain Corollary 1 (a). Corollary 1 (b) follows from an easy reduction from the satisfiability problem. \Box

In fact, any EMSO-definable MSC language is realizable as an MPA and, vice versa, any MSC language of some MPA has an appropriate EMSO counterpart.

Theorem 1. MPA = EMSO

The proof will be based on the concept of *graph acceptors* [16], a generalization of finite automata to labeled graphs, which are known to be expressively equivalent to existential monadic second-order logic wrt. graphs of bounded degree. We consider graph acceptors running on MSCs, thus, on structures of bounded degree², which makes them applicable to our setting. A graph acceptor works on a graph as follows: It first assigns to each node one of its control states and then checks if the local neighborhood of each node (incorporating the state assignment) corresponds to a pattern from a finite supply of so-called *spheres*. In our setting, such a pattern is a labeled graph. For an alphabet Σ , we assume in the following a Σ -labeled graph to be a nonempty and finite structure $(E, \{\leq_p\}_{p \in \mathcal{P}}, <_c, \lambda)$ of degree at most 3. In particular, λ is a mapping $E \to \Sigma$, while the edges can be considered to be $(\mathcal{P} \uplus \{c\})$ -labeled. Note that an MSC is an *Act*-labeled graph, while the converse does not necessarily hold.

Let us become more concrete and let Σ and R be an alphabet and a natural, respectively. Given a Σ -labeled graph $G = (E, \{ \leq_p \}_{p \in \mathcal{P}}, <_c, \lambda)$ (let in the following \prec denote $<_c \cup \bigcup_{p \in \mathcal{P}} <_p)$ and elements $e, e' \in E$, the distance $d_G(e', e)$ from e' to e is ∞ if it holds $(e, e') \notin (\prec \cup \prec^{-1})^*$ and, otherwise, the minimal natural number k such that there is a sequence of elements $e_0, \ldots, e_k \in E$ with $e_0 = e, e_k = e'$, and $e_i \prec e_{i+1}$ or $e_{i+1} \prec e_i$ for each $i \in \{0, \ldots, k-1\}$. Sometimes, if it is clear from the context, we omit the subscript G. An R-sphere over Σ is a Σ -labeled graph $H = (E, \{\leq_p\}_{p \in \mathcal{P}}, <_c, \lambda, \gamma)$ together with a designated sphere center $\gamma \in E$ such that, for any $e \in E, d_H(e, \gamma) \leq R$. Two 2-spheres are shown in Figure 1 where the sphere centers are depicted as rectangles. For a Σ -labeled graph $G = (E, \{\leq_p\}_{p \in \mathcal{P}}, <_c, \lambda)$ and $e \in E$, let the R-sphere of G around e be given by $(E', \{<'_p\}_{p \in \mathcal{P}}, <'_c, \lambda', e)$ where $E' = \{e' \in E \mid d_G(e', e) \leq R\}, <'_p = <_p \cap (E' \times E')$ for each $p \in \mathcal{P}, <'_c = <_c \cap (E' \times E')$, and λ' is the restriction of λ to E'.

² Any node of the *graph* of an MSC has at most three direct neighbors.

152 B. Bollig and M. Leucker

A graph acceptor (over Act) is a structure $\mathcal{GA} = (Q, R, \mathfrak{S}, Occ)$ such that Qis a nonempty finite set of (control) states, $R \in \mathbb{N}$, \mathfrak{S} is a finite set of R-spheres over $Act \times Q$ (as we identify isomorphic structures, we actually deal with a finite set of isomorphism classes), and Occ is a Boolean combination of conditions of the form "sphere $H \in \mathfrak{S}$ occurs at least $\geq n$ times" where $n \in \mathbb{N}$. A run of \mathcal{GA} on an Act-labeled graph $(E, \{\leq_p\}_{p \in \mathcal{P}}, <_c, \lambda)$ is a mapping $\rho : E \to Q$ such that, for each $e \in E$, the R-sphere of $(E, \{<_p\}_{p \in \mathcal{P}}, <_c, (\lambda, \rho))$ around e is isomorphic to some $H \in \mathfrak{S}$. We call ρ accepting if it satisfies the constraints imposed by Occ. The language of \mathcal{GA} wrt. a class \mathcal{K} of Act-labeled graphs, denoted by $L_{\mathcal{K}}(\mathcal{GA})$, is the set of Act-labeled graphs $G \in \mathcal{K}$ on which there is an accepting run of \mathcal{GA} .



Fig. 1. The sphere(s) of a graph acceptor

The rest of this section is dedicated to the proof of Theorem 1.

Proof. It remains to show inclusion from right to left. So let φ be an EMSO sentence. We can assume the existence of a graph acceptor \mathcal{GA} over *Act* that, running on MSCs, recognizes the MSC language defined by φ . In turn, \mathcal{GA} will be translated into an MPA \mathcal{A} that captures the application of \mathcal{GA} to MSCs, i.e., $L(\mathcal{A}) = L_{MSC}(\mathcal{GA})$. So let $\mathcal{GA} = (Q, R, \mathfrak{S}, Occ)$ be a graph acceptor over *Act*.

For our purpose, it suffices to consider only those *R*-spheres $H \in \mathfrak{S}$ for which there is an *extended* MSC $M = (E, \{\leq_p\}_{p \in \mathcal{P}}, <_c, \lambda)$, which has an extended labeling function $\lambda : E \to Act \times Q$, and an event $e \in E$ such that *H* is the *R*sphere of *M* around *e*. Other spheres cannot contribute to an MSC. Because, to become part of a run on some MSC *M*, an *R*-sphere has to admit an embedding into *M*. In this sense, the 2-sphere illustrated in Figure 1 (a) may contribute to a run on an MSC (it can be complemented by a 1!3-labeled event arranged in order between the two other events of process 1), while the 2-sphere illustrated aside is irrelevant and will be ignored in the following. This assumption is essential, as it ensures that, for each $H = (E, \{\leq_p\}_{p \in \mathcal{P}}, <_c, \lambda, \gamma) \in \mathfrak{S}$ and $e \in E$, $d_H(e, \gamma) < R$ implies that *E* also contains a communication partner of *e* wrt. $<_c$.

In the following, we use notions that we have introduced for MSCs also for spheres $(E, \{\leq_p\}_{p \in \mathcal{P}}, \leq_c, \lambda, \gamma)$ over $Act \times Q$, such as P(e), E_p , and \leq_p (to indicate the process of $e \in E$ and as abbreviations for $\lambda^{-1}(Act_p \times Q)$ and the

reflexive transitive closure of \leq_p , respectively)³. For example, considering the 2-sphere from Figure 1 (a), P(a) = 1, $E_1 = \{a, e\}$, and $b \leq_2 d$, but not $a \leq_1 e$. Let $maxE := max\{|E| \mid (E, \{\leq_p\}_{p \in \mathcal{P}}, <_c, \lambda, \gamma) \in \mathfrak{S}\}$ and let \mathfrak{S}^+ be the set of exntended R-spheres, i.e., the set of structures $((E, \{\leq_p\}_{p \in \mathcal{P}}, <_c, \lambda, \gamma, e), i)$ where $(E, \{\leq_p\}_{p \in \mathcal{P}}, <_c, \lambda, \gamma) \in \mathfrak{S}, e \in E$ is the active node, and $i \in \{1, \ldots, 4 \cdot maxE^2 + 1\}$ is the current instance. For $p \in \mathcal{P}$, we define $\mathfrak{S}_p := \{(E, \{\leq_p\}_{p \in \mathcal{P}}, <_c, \lambda, \gamma) \in \mathfrak{S}^+ \mid P(e) = p\}$ and, furthermore, $\mathfrak{S}_p^+ := \{((E, \{\leq_p\}_{p \in \mathcal{P}}, <_c, \lambda, \gamma, e), i) \in \mathfrak{S}^+ \mid P(e) = p\}$. Finally, let max(Occ) denote the least threshold n such that Occ does not distinguish occurrence numbers $\geq n$.

For readability, we let in the following \prec denote the collection of relations $(\{ \leq_p \}_{p \in \mathcal{P}}, <_c)$ and just write $(E, \prec, \lambda, \gamma)$ instead of $(E, \{ \leq_p \}_{p \in \mathcal{P}}, <_c, \lambda, \gamma)$.

The idea of the transformation is that, roughly speaking, A guesses a tiling of the MSC to be read and then verifies that the tiling corresponds to an accepting run of \mathcal{GA} . Accordingly, a local state of \mathcal{A} holds a set of *active R*-spheres, i.e., a set of spheres that play a role in its immediate environment of distance at most R. Each local state s (apart from the initial states, as we will see) carries exactly one extended *R*-sphere $((E, \prec, \lambda, \gamma, e), i) \in \mathfrak{S}^+$ with $\gamma = e$, which means that a run of \mathcal{GA} assigns $(E, \prec, \lambda, \gamma)$ to the event that corresponds to s. To establish isomorphism between $(E, \prec, \lambda, \gamma)$ and the *R*-sphere induced by *s*, *s* transfers/obtains its obligations in form of an extended R-sphere $((E, \prec, \lambda, \gamma, e'), i)$ to/from its immediate neighbors, respectively. For example, provided e is labeled with a send action and there is $e' \in E$ with $e <_{c} e'$, the message to be sent in state s will contain $((E, \prec, \lambda, \gamma, e'), i)$, which, in turn, the receiving process understands as a requirement to be satisfied. As there may be an overlapping of isomorphic *R*-spheres, a state can hold several instances of one and the same sphere, which then refer to distinct states/events as corresponding sphere center. Those instances will be distinguished by means of the natural i. The benefit of i will become clear before long.

Let us turn to the construction of $\mathcal{A} = ((\mathcal{A}_p)_{p \in \mathcal{P}}, \mathcal{D}, \overline{s}^{in}, F), \mathcal{A}_p = (S_p, \Delta_p)$, which is given as follows: For $p \in \mathcal{P}$, a local state of \mathcal{A}_p is a pair (\mathcal{S}, ν) where

- $-\nu$ is a mapping $\mathfrak{S}_p \to \{0, \dots, \max(Occ)\}$ (let in the following ν_p^0 denote the function that maps each *R*-sphere $H \in \mathfrak{S}_p$ to 0) and
- $-\mathcal{S}$ is either the empty set or it is a subset of \mathfrak{S}_p^+ such that
 - there is exactly one extended *R*-sphere $((\vec{E}, \prec, \lambda, \gamma, e), i) \in S$ with $\gamma = e$ (whose component $(E, \prec, \lambda, \gamma)$ we identify by $\varsigma(S)$ from now on) and
 - for any two $((E,\prec,\lambda,\gamma,e),i), ((E',\prec',\lambda',\gamma',e'),i') \in S$,
 - (a) $\lambda(e) = \lambda'(e') \in Act_p \times Q$ (so that we can assign a well-defined unique label $\lambda(S) \in Act_p \times Q$ to S, namely the labeling $\lambda(e)$ for some $((E, \prec, \lambda, \gamma, e), i) \in S$) and
 - (b) if $(E, \prec, \lambda, \gamma) \cong (E', \prec', \lambda', \gamma')$ and i = i', then e = e'.

The set \mathcal{D} of synchronization messages is the cartesian product $2^{\mathfrak{S}^+} \times 2^{\mathfrak{S}^+}$. Roughly speaking, the first component of a message contains obligations the re-

³ Note that, wrt. spheres, $\leq_{\mathbf{p}}$ is not necessarily a total order.

ceiving state/event has to satisfy, while the second component imposes requirements that must *not* be satisfied by the receiving process to ensure isomorphism. We now turn towards the definition of Δ_p and define $((S, \nu), \sigma, (\mathcal{P}, \mathcal{N}), (S', \nu')) \in \Delta_p$ if the following hold:

- 1. $\lambda(\mathcal{S}') = (\sigma, s)$ for some $s \in Q$.
- 2. For any $((E, \prec, \lambda, \gamma, e), i) \in S$ and $e' \in E_p$, if $((E, \prec, \lambda, \gamma, e'), i) \in S'$, then $e <_p e'$.
- 3. For any $((E, \prec, \lambda, \gamma, e), i) \in S'$, if $S \neq \emptyset$ and e is minimal in (E_p, \leq_p) , then $d(e, \gamma) = R$.
- 4. For any $((E, \prec, \lambda, \gamma, e), i) \in S$, if e is maximal in (E_p, \leq_p) , then $d(e, \gamma) = R$.
- 5. For any $((E, \prec, \lambda, \gamma, e), i) \in S'$, if e is not minimal in (\overline{E}_p, \leq_p) , then we have $((E, \prec, \lambda, \gamma, e^-), i) \in S$ where $e^- \in E_p$ is the unique event with $e^- <_p e$.
- For any ((E, ≺, λ, γ, e), i) ∈ S, if e is not maximal in (E_p, ≤_p), then we have ((E, ≺, λ, γ, e⁺), i) ∈ S' where e⁺ ∈ E_p is the unique event such that e ≤_p e⁺.
- 7. (i) In case that $\sigma = p!q$ for some $q \in \mathcal{P}$:
 - (a) for any ((E, ≺, λ, γ, e), i) ∈ S' and any e' ∈ E, if e <_c e', then we have ((E, ≺, λ, γ, e'), i) ∈ P,
 - (b) for any ((E, ≺, λ, γ, e), i) ∈ S' and any e' ∈ E, if e ∠_c e', then we have ((E, ≺, λ, γ, e'), i) ∈ N, and
 - (c) for any $((E, \prec, \lambda, \gamma, e), i) \in \mathcal{P}$, there is $e' \in E$ such that $e' <_{c} e$ and $((E, \prec, \lambda, \gamma, e'), i) \in S'$.
 - (ii) In case that $\sigma = p?q$ for some $q \in \mathcal{P}$:
 - (a) $\mathcal{P} \subseteq \mathcal{S}'$,
 - (b) $\mathcal{N} \cap \mathcal{S}' = \emptyset$, and
 - (c) for any $((E, \prec, \lambda, \gamma, e'), i) \in S'$, if there is $e \in E$ with $e <_{c} e'$, then $((E, \prec, \lambda, \gamma, e'), i) \in \mathcal{P}$.
- 8. $\nu' = \nu[\varsigma(\mathcal{S}')/\min\{\nu(\varsigma(\mathcal{S}')) + 1, \max(Occ)\}]$ (i.e., ν' maps $\varsigma(\mathcal{S}')$ to the minimum of $\nu(\varsigma(\mathcal{S}')) + 1$ and $\max(Occ)$ and, otherwise, coincides with ν).

Thus, Condition 1. guarantees that any state within a run has the same labeling as the event it is assigned to. Condition 2. makes sure that, whenever there is a $<_p$ -edge in the input MSC, then there is a corresponding edge in the extended sphere that is passed from the source to the target state of the corresponding transition. Conversely, if there is no $<_p$ -edge between two nodes in the extended sphere, then it must not be passed directly to impose the same behavior on the MSC, i.e., the corresponding events in the MSC must not touch each other. Conditions 3. and, dually, 4. make sure that a sphere that does not make use of the whole radius *R* is employed in the initial or final phase of a run only. By Conditions 5. and 6., extended spheres must be passed along a process line as far as possible, hereby starting in a minimal and ending in a maximal active node. Condition 7. ensures the corresponding beyond process lines, i.e., for messages. Finally, Condition 8. guarantees that the second component of each state correctly keeps track the number of spheres used so far.

Furthermore, $\overline{s}^{in} = ((\emptyset, \nu_p^0))_{p \in \mathcal{P}}$ and, for $(\mathcal{S}_p, \nu_p) \in \mathcal{S}_p$, $((\mathcal{S}_p, \nu_p))_{p \in \mathcal{P}} \in F$ if the union of mappings ν_p satisfies the requirements imposed by *Occ* and, for all

 $p \in \mathcal{P}$ and $((E, \prec, \lambda, \gamma, e), i) \in \mathcal{S}_p$, e is maximal in (E_p, \leq_p) . In fact, it holds $L(\mathcal{A}) = L_{MSC}(\mathcal{GA})$.

Let $\rho: \widetilde{E} \to Q$ be an accepting run of \mathcal{GA} on $M = (\widetilde{E}, \{\widetilde{\prec}_p\}_{p \in \mathcal{P}}, \widetilde{\prec}_c, \widetilde{\lambda}) \in \mathbb{MSC}$ and let $\widehat{\rho}$ denote the mapping $\widetilde{E} \to \mathfrak{S}$ that maps an event $e \in \widetilde{E}$ onto the *R*-sphere of $(\widetilde{E}, \{\widetilde{\prec}_p\}_{p \in \mathcal{P}}, \widetilde{\prec}_c, (\widetilde{\lambda}, \rho))$ around *e*. In an accepting run (r, m) of \mathcal{A} on *M*, *r* basically assigns to an event $e \in \widetilde{E}$ —apart from the obvious mapping ν —the set of those extended spheres $((E, \prec, \lambda, \gamma, e_0), i) \in \mathfrak{S}^+$ such that there is an event $e' \in \widetilde{E}$ with both $d_M(e', e) \leq R$ and $(E, \prec, \lambda, \gamma, e_0)$ is isomorphic to $(\widehat{\rho}(e'), e)$. Hereby, max*E* is sufficiently large to guarantee an instance labeling that is consistent with the transition relation of \mathcal{A} . If we suppose $m: \widetilde{\prec}_c \to \mathcal{D}$ to map a pair $(e_s, e_r) \in \widetilde{\prec}_c$ onto $(\mathcal{P}, \mathcal{N})$ where (set (\mathcal{S}, ν) to be $r(e_s)$) $\mathcal{P} = \{((E, \prec, \lambda, \gamma, e_0'), i) \in \mathfrak{S}^+ |$ there is $e_0 \in E$ with $((E, \prec, \lambda, \gamma, e_0), i) \in \mathcal{S}$ and $e_0 \prec_c e'_0\}$ and $\mathcal{N} = \{((E, \prec, \lambda, \gamma, e'_0), i) \in \mathcal{S} \text{ and } e_0 \notin_c e'_0\}, (r, m)$ is an accepting run of \mathcal{A} on M.

Conversely, let (r, m) be an accepting run of \mathcal{A} on $M = (E, \{\leq_p\}_{p \in \mathcal{P}}, <_c, \lambda) \in \mathbb{MSC}$. If we define $\rho : E \to Q$ to map an event $e \in E$ to the control state that is associated with the sphere center of $\varsigma(\mathcal{S})$ where $r(e) = (\mathcal{S}, \nu)$ for some ν , then ρ turns out to be an accepting run of \mathcal{GA} on M.

Example 1. In the following, let *H* denote the 2-sphere from Figure 1 (a). Figure 2, showing some MSC M with four processes, illustrates the transition behavior of the MPA \mathcal{A} from the above proof. It demonstrates how a run of \mathcal{A} on M transfers extensions of H from one event of M to a neighboring one to make sure that the 2-sphere around event e_c (which is indicated by solid edges) is isomorphic to H. For example, the state that is taken on event e_a may contain the extended sphere (H, a). (For clarity, control states and the natural *i* to distinguish different instances of spheres are omitted.) As a $<_{c}$ b (wrt. the edge relation of H), A passes (H,b) in form of a message to process 2. Receiving (H, b), process 2 becomes aware it should bind e_b to some state that contains (H, b) (conditions 7. (i) (a) and 7. (ii) (a) from the definition of the transition relation). As, in H, b is followed by c, so e_c has to be associated with a state containing (H,c) (condition 6.). In contrast, $e_{\mathbf{h}}$ is not allowed to carry the extended sphere (H, e), unless it belongs to a different instance of H (condition 2.). Now consider e_d , which holds the extended sphere (H, d). Due to condition 5., the preceding state, which is associated to e_c , must contain (H,c), which means that a run cannot simply enter H beginning with d. Moreover, as e_d is a receive event, A has to receive a message containing (H, d) (condition 7. (ii) (c)). In turn, the corresponding send event e_e has to be associated with a state that holds (H,e) (condition 7. (i) (c)). Note that, as d(a,c) = d(e,c) = 2, the (illustrated parts of the) states assigned to e_a and e_e satisfy conditions 3. and 4.

5 Beyond Realizability

In this section, we show that MSO logic over MSCs is strictly more expressive than EMSO. Together with the results of the previous section, this will be used



Fig. 2. Simulating a graph acceptor

to show that MPAs cannot be complemented in general. More specifically, we show that quantifier alternation forms a hierarchy:

Theorem 2. The monadic quantifier-alternation hierarchy over **MSC** is infinite.

Proof. Matz and Thomas proved infinity of the monadic quantifier-alternation hierarchy over *grids* [11,16]. Using an idea from [15], we show how grids can be encoded into MSCs and then rewrite their result in terms of MSCs adapting their proof to our setting.

For a positive natural $n \in \mathbb{N}_{\geq 1}$, we use [n] as a shorthand for $\{1, \ldots, n\}$. Given $n, m \in \mathbb{N}_{\geq 1}$, the (n, m)-grid (with n rows and m columns) is the structure $g(n,m) := ([n] \times [m], S_1, S_2)$ where $S_1, S_2 \subseteq ([n] \times [m])^2$ contain the pairs $((i, j), (i + 1, j)) \in ([n] \times [m])^2$ and $((i, j), (i, j + 1)) \in ([n] \times [m])^2$, respectively. A relation $R \subseteq \mathbb{N}_{\geq 1} \times \mathbb{N}_{\geq 1}$ may be represented by the grid language $\{g(n,m) \mid (n,m) \in R\}$. As a unary function $f : \mathbb{N}_{\geq 1} \to \mathbb{N}_{\geq 1}$ can be considered as a binary relation, we define the grid language $\mathcal{G}(f)$ of f to be the set $\{g(n, f(n)) \mid n \in \mathbb{N}_{\geq 1}\}$. A grid g(n,m) can be folded to an MSC M(n,m) as exemplarily shown for g(3, 5) in Figure 3.



Fig. 3. Folding the (3,5)-grid

A similar encoding is used by Kuske to prove infinity of the monadic quantifieralternation hierarchy for certain *pomsets* over at least two processes [8]. However, we introduce a third process to obtain distinguished labelings of events that mark the end of a column in the grid to be encoded, which is signalized by sending a message to process 3. By the type of an event, we furthermore recognize which events really correspond to a node of the grid, namely those that are labeled with a send action performed by process 1 or 2.

A grid language \mathcal{G} defines the MSC language $L(\mathcal{G}) := \{M(n,m) \mid g(n,m) \in \mathcal{G}\}$. For a function $f : \mathbb{N}_{\geq 1} \to \mathbb{N}_{\geq 1}$, we furthermore write L(f) as a shorthand for the MSC language $L(\mathcal{G}(f))$. We now closely follow [16], which resumes the result of [11]. So let, for $k \in \mathbb{N}$, the functions $s_k, f_k : \mathbb{N}_{\geq 1} \to \mathbb{N}_{\geq 1}$ be inductively defined via $s_0(n) = n, s_{k+1}(n) = 2^{s_k(n)}, f_0(n) = n$, and $f_{k+1}(n) = f_k(n) \cdot 2^{f_k(n)}$.

Claim 1. For each $k \in \mathbb{N}$, the MSC language $L(f_k)$ is Σ_{2k+3} -definable.

Proof of Claim 1. It is easy to prove that the set of possible grid foldings is EMSO-definable (or, equivalently, the language of some MPA). As, furthermore, a grid is interpretable in a grid folding by first-order formulas, we can show that, for any $k \ge 1$, if a grid language \mathcal{G} is Σ_k -definable (over grids), then $L(\mathcal{G})$ is Σ_k -

definable (over MSCs). The claim follows from the fact that any grid language $\mathcal{G}(f_k)$ is Σ_{2k+3} -definable [16].

Claim 2. Let $f : \mathbb{N}_{\geq 1} \to \mathbb{N}_{\geq 1}$ be a function. If L(f) is Σ_k -definable (over MSCs) for some $k \geq 1$, then f(n) is in $s_k(\mathcal{O}(n))$.

Proof of Claim 2. Let $k \ge 1$ and let in the following the events of an MSC $(E, \{\leq_p\}_{p \in \mathcal{P}}, <_{c}, \lambda)$ be labeled with elements from $Act \times \{0, 1\}^i$ for some $i \in \mathcal{P}$ $\mathbb{N}_{>1}$, i.e., $\lambda: E \to Act \times \{0,1\}^i$. But note that the type of an event still depends on the type of its communication action only. Let furthermore $\varphi(Y_1, \ldots, Y_i)$ be a Σ_k -formula defining a set of MSCs over the new label alphabet that are foldings of grids. For a fixed column length $n \ge 1$, we will build a nondeterministic finite word automaton \mathcal{A}_n over $(Act \times \{0,1\}^i)^n$ with $s_{k-1}(c^n)$ states (for some constant c) that reads grid-folding MSCs column by column and is equivalent to $\varphi(Y_1,\ldots,Y_i)$ wrt. grid foldings with column length n. Column here means a sequence of communication actions, each provided with an additional label, that represents a column in the corresponding grid. For example, running on the MSC M(3,5) as shown in Figure 3, A_3 first reads the letter $[(1!2)^2(1!3)(3?1)(3!2)]$ (recall that each action is still provided with an extra labeling, which we omit here for the sake of clarity), then continues reading $[((2?1)(2!1))^2(2?3)(2!3)(3?2)(3!1)]$ and so on. Then the shortest word accepted by \mathcal{A}_n has length $\leq s_{k-1}(c^n)$ so that, if $\varphi(Y_1, \ldots, Y_i)$ defines an MSC language L(f) for some f, we have $f(n) \in s_k(\mathcal{O}(n))$. Let us now turn to the construction of \mathcal{A}_n . The formula $\varphi(Y_1,\ldots,Y_i)$ is of the form $\exists \overline{X_k} \forall \overline{X_{k-1}} \ldots \exists / \forall \overline{X_1} \psi(Y_1,\ldots,Y_i,\overline{X_k},\ldots,\overline{X_1})$ or, equivalently, $\exists \overline{X_k} \neg \exists \overline{X_{k-1}} \dots \neg \exists \overline{X_1} \psi'(Y_1, \dots, Y_i, \overline{X_k}, \dots, \overline{X_1})$. We proceed by induction on k. For $k = 1, \varphi(Y_1, \dots, Y_i)$ is an EMSO formula. According to [16], its MSC language (consisting of MSCs with extended labelings) coincides with the MSC language of some graph acceptor. The transformation from graph acceptors to MPAs from the proof of Theorem 1 can be easily adapted to handle the extended labeling. Thus, $\varphi(Y_1, \ldots, Y_i)$ defines a language that is realizable by an MPA $\mathcal{A} = ((\mathcal{A}_p)_{p \in \mathcal{P}}, \mathcal{D}, \overline{s}^{in}, F)$. The automaton \mathcal{A}_n can now be obtained from \mathcal{A} using a part of its global transition relation $\Longrightarrow_{\mathcal{A}} \subseteq (S_{\mathcal{A}} \times C_{\mathcal{A}}) \times ((Act \times \{0,1\}^i) \times C_{\mathcal{A}})$ \mathcal{D} × ($S_A \times \mathcal{C}_A$) (as it is defined, for example, in [6]) where S_A is the cartesian product of the local state spaces of \mathcal{A} and $\mathcal{C}_{\mathcal{A}} := \{\chi \mid \chi : Ch \to (\mathcal{D} \uplus \{\bot\})^n\}$ is the set of possible channel contents. Note that only a bounded number of channel contents has to be considered, as the set of grid foldings with column length n forms a max $\{1, n-1\}$ -bounded MSC language (cf. [6] for the definition of boundedness). Due to $|S_A \times C_A| \leq (|S_A| \cdot (|\mathcal{D}|+1))^{|Ch| \cdot n} \leq c^n$ for some constant $c, c^n = s_0(c^n)$ is an upper bound for the number of states of \mathcal{A}_n , which only depends on the automaton \mathcal{A} and, thus, on $\varphi(Y_1, \ldots, Y_i)$. The induction steps respectively involve both a complementation step (for negation) and a projection step (concerning existential quantification). While the former increases the number of states exponentially, the latter leaves it constant so that, altogether, the required number of states is obtained. This concludes the proof of Claim 2.

As $f_{k+1}(n)$ is not in $s_k(\mathcal{O}(n))$, it follows from Claims 1 and 2 that the hierarchy of classes of Σ_k -definable MSC languages (k = 1, 2, ...) is infinite.

Corollary 2. $MPA \subsetneq MSO$

As $\mathcal{MPA} = \mathcal{EMSO}$, it follows that the *complement* $\overline{L} := \{M \in \mathbb{MSC} \mid M \notin L\}$ of an MSC language $L \in \mathcal{MPA}$, is not necessarily contained in \mathcal{MPA} , too [15]. Thus, we get the answer to an open question, which has been raised by Kuske [9].

Theorem 3. MPA is not closed under complement.

6 Discussion

Recall that we consider an MSC to be a graph, which corresponds to the view taken in [10] but is different from the one in [6,9], who model an MSC as a labeled partial order (E, \leq, λ) . However, while the way to define an MSC immediately affects the syntax and expressivity of (fragments of) the corresponding monadic second-order logic, Theorem 3 holds independently of that modeling, for the following reason: there is a one-to-one-correspondence between an MSC structure $(E, \{ \leq_p \}_{p \in \mathcal{P}}, <_c, \lambda)$ and its counterpart (E, \leq, λ) with $\leq = (<_c \cup \bigcup_{p \in \mathcal{P}} <_p)^*$. This correspondence carries over to MSO logic in the signature proposed in this paper. In other words, an MSO formula is satisfied by $(E, \{ <_p \}_{p \in \mathcal{P}}, <_c, \lambda)$ iff it is satisfied by (E, \leq, λ) (where a formula will be interpreted over labeled partial orders (E, \leq, λ) of MSCs in the obvious manner). As the definition of a message-passing automaton is robust against the concrete modeling, too, Theorem 3 can be applied to any common definition of what an MSC is. However, our logic can only be considered to be the canonical (existential) monadic second-order logic if MSCs are given by their graphs.

If, for some $B \ge 1$, we restrict to *B*-bounded MSCs (see [6] for details), EMSO[$\leq_p, <_c$], MSO[$\leq_p, <_c$], EMSO[\leq], and MSO[\leq] coincide wrt. expressiveness. Thus, our work subsumes the work by Henriksen et al. [6].

Note that, for clarity, an MSC does not carry any information about the concrete messages to be sent. However, preceding results can be easily extended towards MSCs that are equipped with message information, as they are provided in [1,2,5], for example.

Let us recall the results of the previous sections: We have studied the class of MSC languages that corresponds to EMSO logic and MPAs. By means of graph acceptors, we have shown that MPAs are expressively equivalent to EMSO logic. In particular, for every EMSO sentence, there exists an equivalent MPA. Our proof is based on results by Thomas, which, in turn, refer to Hanf's Theorem. For practical applications, it would be desirable to have a simple effective transformation from (fragments of) EMSO to MPAs of reasonable complexity.

Furthermore, we proved that the class of MSC languages definable in MSO logic is strictly larger. Consequently, MPAs cannot be complemented in general. This question was raised in [9].

It remains to discuss the relation between the nondeterministic automata model with a deterministic one in the unbounded setting. In [13,9], it was shown

that deterministic MPAs suffice to realize regular bounded MSC languages. This question was also addressed in [4] regarding the related model of asynchronous cellular automata for pomsets without autoconcurrency.

It would also be interesting to have logics that capture formalisms such as locally- and globally-synchronized HMSCs and related automata models [5].

References

- 1. R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In 22nd International Conference on Software Engineering. ACM, 2000.
- R. Alur and M. Yannakakis. Model checking of message sequence charts. In CONCUR 1999, volume 1664 of LNCS. Springer, 1999.
- 3. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2), 1983.
- 4. M. Droste, P. Gastin, and D. Kuske. Asynchronous cellular automata for pomsets. *Theoretical Computer Science*, 247(1–2), 2000.
- B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun. Infinite-state high-level MSCs: Model-checking and realizability. In *ICALP 2002*, volume 2380 of *LNCS*. Springer, 2002.
- J. G. Henriksen, M. Mukund, K. Narayan Kumar, and P. S. Thiagarajan. Regular collections of message sequence charts. In *MFCS 2000*, volume 1893 of *LNCS*. Springer, 2000.
- 7. ITU-TS. ITU-TS Recommendation Z.120: Message Sequence Chart 1999 (MSC99). Technical report, ITU-TS, Geneva, 1999.
- 8. D. Kuske. Asynchronous cellular automata and asynchronous automata for pomsets. In *CONCUR 1998*, volume 1466 of *LNCS*, 1998.
- 9. D. Kuske. Regular sets of infinite message sequence charts. *Information and Computation*, 187:80–109, 2003.
- 10. P. Madhusudan. Reasoning about sequential and branching behaviours of message sequence graphs. In *ICALP 2000*, volume 2076 of *LNCS*. Springer, 2001.
- 11. O. Matz and W. Thomas. The monadic quantifier alternation hierarchy over graphs is infinite. In *LICS 1997*. IEEE Computer Society Press, 1997.
- R. Morin. Recognizable sets of message sequence charts. In STACS 2002, volume 2285 of LNCS. Springer, 2002.
- M. Mukund, K. Narayan Kumar, and M. Sohoni. Synthesizing distributed finitestate systems from MSCs. In *CONCUR 2000*, volume 1877 of *LNCS*. Springer, 2000.
- 14. A. Muscholl and D. Peled. Message sequence graphs and decision problems on Mazurkiewicz traces. In *MFCS 1999*, volume 1672 of *LNCS*. Springer, 1999.
- 15. W. Thomas. Elements of an automata theory over partial orders. In *POMIV 1996*, volume 29 of *DIMACS*. AMS, 1996.
- 16. W. Thomas. Automata theory on trees and partial orders. In *TAPSOFT 1997*, volume 1214 of *LNCS*. Springer, 1997.
- W. Thomas. Languages, automata and logic. In A. Salomaa and G. Rozenberg, editors, *Handbook of Formal Languages*, volume 3, Beyond Words. Springer, Berlin, 1997.

Symbolic Bisimulation in the Spi Calculus*

Johannes Borgström, Sébastien Briais, and Uwe Nestmann

School of Computer and Communication Sciences EPFL-I&C, 1015 Lausanne, Switzerland

Abstract. The spi calculus is an executable model for the description and analysis of cryptographic protocols. Security objectives like secrecy and authenticity can be formulated as equations between spi calculus terms, where equality is interpreted as a contextual equivalence.

One problem with verifying contextual equivalences for messagepassing process calculi is the infinite branching on process input. In this paper, we propose a general symbolic semantics for the spi calculus, where an input prefix gives rise to only one transition.

To avoid infinite quantification over contexts, non-contextual concrete bisimulations approximating barbed equivalence have been defined. We propose a symbolic bisimulation that is sound with respect to barbed equivalence, and brings us closer to automated bisimulation checks.

1 Background, Related Work, and Summary

Verification of Cryptographic Protocols in the Spi Calculus. Abadi and Gordon designed the spi calculus as an extension of the pi calculus with encryption primitives in order to describe and formally analyze cryptographic protocols [AG99]. The success of the spi calculus is due to at least three reasons. (1) It is equipped with an operational semantics; thus any protocol described in the calculus may be regarded as executable. (2) Security properties can be formulated as equations on process terms, so no external formalism is needed. (3) Contextual equivalences on process terms avoid the need to explicitly model the attacker; they take into account any attacker that can be expressed in the calculus.

For example, we may wish to analyze the trivial cryptographic protocol

$$(
u k) \left(A \,|\, B
ight) \quad ext{where} \quad A := \overline{a} \langle \mathsf{E}_k m
angle \quad ext{and} \quad B := a(x). \overline{f} \langle \mathsf{D}_k x
angle$$

consisting of participant A sending on channel a the message m, encrypted under the secret shared symmetric key k, to participant B who tries to decrypt the received message and, in case of successful decryption, outputs the result on channel f. We may compare this protocol with its *specification*

$$(\nu \underline{k}) (\underline{A} \,|\, \underline{B}) \quad ext{where} \quad \underline{A} := \overline{a} \langle \mathsf{E}_{\underline{k}} m
angle \quad ext{and} \quad \underline{B} := a(y). [\mathsf{D}_{\underline{k}} y : \mathcal{M}] \overline{f} \langle m
angle$$

where \underline{B} transmits the correct message m on channel f whenever the dummy message (on reception bound to y) can be decrypted (as expressed by the guard

^{*} Supported by the Swiss National Science Foundation, grant No. 21-65180.1.

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 161-176, 2004.



Fig. 1. Equivalences

 $[D_{\underline{k}}y:\mathcal{M}]$). If the equation $(\nu \underline{k})(\underline{A} | \underline{B}) = (\nu k)(A | B)$ holds, then no context is able to influence the authenticity (more precisely: integrity) of the message *m*.

Apart from the equational style, cryptographic protocols in the spi calculus are analyzed by control flow analysis, trace analysis, reachability analysis, and type systems; they are beyond the scope of this paper.

Equivalences. To verify security properties expressed in the equational style, we need to give an interpretation for the equation symbol. *Contextual* equivalences—two terms are related if they behave in the same way *in all contexts*—are attractive because the quantification over all contexts directly captures the intuition of an unknown attacker expressible within the spi calculus [AG99].

The notions of may-testing equivalence and barbed equivalence are the most prominent contextual equivalences [see the right column of Fig. 1]. Their main distinction is linear time versus branching time: The former considers the possibility of passing tests after sequences of computation steps; the latter has a more refined view, also comparing the derivatives of internal computation. Secrecy and authenticity are usually seen as trace-based properties and formulated in terms of testing equivalence; however, testing is not known to be sufficient for anonymity or fairness [CS02].

Proof Methods for Contextual Equivalences. Although intuitive, the quantification over contexts makes direct proofs of contextual equivalences notoriously difficult. This problem is traditionally dealt with by defining equivalent noncontextual relations [see the middle column of Fig. 1]. Applying this pattern to the spi calculus, Boreale, De Nicola, and Pugliese [BDP02] introduced a *trace equivalence* corresponding to testing equivalence, as well as an "environmentsensitive" *labeled bisimulation* as the counterpart of barbed equivalence.

Because of the practical usefulness of the definition of bisimulations in terms of co-induction, they are used as proof techniques for trace-based equivalences. With this goal, and in a style quite different to [BDP02], Abadi and Gordon proposed *framed* bisimulation [AG98], that is however incomplete with respect to barbed equivalence. This was analyzed and remedied by Borgström and Nestmann, yielding *hedged bisimulation* [BN02].

Infinite Branching & Symbolic Proof Methods. Once we have provided a noncontextual alternative for our chosen equivalence, we face an inherent problem with the operational semantics of message-passing process calculi: The possibility to receive arbitrary messages (like participant *B* performs along channel *a* in the example above) gives rise to an infinite number of "concrete" transitions. Using a less concrete semantics for process input [HL95, BD96], the substitution of received messages for input variables never takes place. Instead, an input prefix produces a single "symbolic" transition, where the input variable is instantiated *lazily*, i.e., only when used, and *indirectly* by collecting the constraints on it that are necessary for a transition to take place. This idea was exploited to implement bisimulation-checking algorithms for the pi calculus [San96, VM94].

Symbolic semantics have also been defined for the limited setting of nonmobile spi calculi, where no channel-passing is allowed or channels do not even exist: examples are the works by Huima [Hui99], Boreale [Bor01], Amadio and Lugiez [AL00], and Fiore and Abadi [FA01]. For the full spi calculus, where complex messages including keys and channel names pose new challenges, the only symbolic semantics that we are aware of was proposed by Durante et al. [DSV03]. However, it is rather complicated, mainly since it is tailored to capture trace semantics. We seek a simpler and more general symbolic semantics, that should also work well for bisirnulation techniques.

Towards Symbolic Bisimulation. In this paper, we propose a symbolic bisimulation for the spi calculus. Here, the elements of a bisimulation consist of a process pair and an environment; the latter captures the knowledge that an attacker has acquired in previous interactions with the process pair. This considerably complicates the generalization of symbolic bisimulation from pi to spi: (1) we must keep track of when an attacker has learned some piece of information so that he can only use it for instantiating inputs taking place later on; (2) the combination of scope extrusion and complex guards and expressions makes a precise correspondence to concrete semantics challenging; (3) the cryptographic knowledge of the environment should be represented clearly and compactly; (4) environment *inconsistency*, signaling that the environment has noticed a difference between the supposedly equivalent processes, must be carefully defined. These challenges are in parts shared with existing work on symbolic trace equivalence [DSV03]. We, however, propose a symbolic bisimulation. For this, hedged bisimulation is a good starting point since it offers a compact and clear knowledge representation.

Contributions of the Paper. We give a general symbolic semantics, not using auxiliary environments, for the full spi calculus. We then use this semantics to define the, to our knowledge, first symbolic bisimilarity for any spi calculus. These tasks are significantly more demanding than a straightforward adaptation of existing approaches in less complex calculi (see the above remarks). We show that this bisimulation is sound with respect to its concrete counterpart, but not complete. We argue that the incompleteness is not problematic for protocol verification, and propose in general terms how it could be removed.

Summary. In §2, we briefly recall the version of the spi calculus that we are using. In §3, we compare the standard concrete operational semantics with a reasonably simple symbolic operational semantics. The latter is used, in §4, as the

foundation for a symbolic "very late" hedged bisimulation, which is then shown to be sound with respect to concrete hedged bisimulation. In §5, we exhibit the proof technique on an example. We highlight, in §6, some incompletenesses that are, however, unproblematic for the security equations that we strive to prove. Conclusions and discussions on future work can be found in §7.

A long version is available via http://lamp.epfl.ch/~jobo/.

2 The Spi Calculus

We assume the reader to have some basic familiarity with the notions and terminology of the pi calculus. Extending the pi calculus, the spi calculus also permits the transmission of complex messages, provided by the addition of primitive constructs for symmetric (shared-key) and asymmetric (public/private-key) encryption ($E_K M$) and decryption ($D_K M$), as well as hashing [AG99, Cor03]. In the long version of this paper, we also have primitive constructs for pairing and pair splitting, generalizing the possibility of the polyadic π -calculus to send several items atomically with nesting under encryption.

We build on the same assumptions on the perfection of the underlying cryptographic system as [AG99, BDP02], which we do not repeat here. As in [AG99, BDP02], and in contrast to [DSV03], we require channels to be names (i.e., not compound messages). This effectively gives the attacker the possibility to verify if a message is a name by attempting to transmit on it.

We assume an infinite set \mathcal{N} of names. Names are used for channels, variables and cleartexts of messages. Hashing and public and private keys are denoted by the operator names $op \in \{H, pub, priv\}$. Expressions F are formed arbitrarily using decryption, encryption and operators; messages M may not contain decryption. Logical formulae ϕ generalize matching with conjunction and negation. The predicate $[F:\mathcal{N}]$ tests for whether F evaluates to a plain name. We also have a (redundant) predicate $[F:\mathcal{M}]$ to check whether the decryptions in a term can be successfully performed. Process constructs include input, output and guard prefixes, parallel composition and restriction.

$a,b,c\ldots,k,l,m,n\ldots,x,y,z$	${f names}~~{\cal N}$
$M, N ::= a \mid E_N M \mid H(M) \mid pub(M) \mid priv(M)$	${\rm messages}\; {\cal M}$
$F, G ::= a \mid E_G F \mid D_G F \mid H(F) \mid pub(F) \mid priv(F)$	expressions ${\cal E}$
$\phi, \psi ::= tt \mid \phi \land \phi \mid \neg \phi \mid [F = G] \mid [F : \mathcal{N}] \mid [F : \mathcal{M}]$	formulae ${\cal F}$
$P,Q ::= 0 F(x).P \overline{F}(G).P \phi P P + P P P$	$(\nu a) P$ processes \mathcal{P}

Free and bound names of terms and sets of terms are inductively defined as expected: *a* is bound in " $(\nu a) P$ " and *x* is bound in "F(x).P". Two processes are α -equivalent if they can be made equal by conflict-free renaming of bound names. We identify α -equivalent processes, except during the derivation of transitions. To treat asymmetric encryption, if F = pub(G) (resp. priv(G)), we define F^{-1} to be priv(G) (resp. pub(G)) and otherwise we let $F^{-1} = F$.

Substitutions σ are partial functions $[F_1/x_1, \ldots, F_n/x_n]$ from names x to expressions F. Substitutions are applied to processes, expressions, formulae and

actions (see below) in the straightforward way, obeying the usual assumption that capture of bound names is avoided by implicit α -conversion: for example, $P\left[\frac{F}{x}\right]$ replaces all free occurrences of x in P by F, renaming bound names in P where needed.

3 Semantics – Concrete and Symbolic

Concrete Semantics. The concrete semantics we use is similar to the one used in [BDP02], apart from that we do not have a *let*-construct in the language. Because of this, input and output forms can contain arbitrary expressions, so we must make sure that these expressions evaluate to a concrete message or channel name before performing the transition. For the *concrete evaluation* of expressions we use a function $\mathbf{e}_{c}(\cdot) : \mathcal{E} \to \mathcal{M} \cup \{\bot\}$.

 $\hat{F} \stackrel{\mathbf{e}_{c}}{\mapsto} \begin{cases} a & \text{if } \hat{F} = a \\ \mathsf{E}_{N}M & \text{if } \hat{F} = \mathsf{E}_{F}G \text{ and } \mathbf{e}_{c}(G) = M \in \mathcal{M} \text{ and } \mathbf{e}_{c}(F) = N \in \mathcal{M} \\ M & \text{if } \hat{F} = \mathsf{D}_{F}G \text{ and } \mathbf{e}_{c}(G) = \mathsf{E}_{N}M \in \mathcal{M} \text{ and } \mathbf{e}_{c}(F) = N^{-1} \\ \mathsf{op}(M) & \text{if } \hat{F} = \mathsf{op}(F) \text{ and } \mathsf{op} \in \{\mathsf{H},\mathsf{pub},\mathsf{priv}\} \text{ and } \mathbf{e}_{c}(F) = M \in \mathcal{M} \\ \bot & \text{if otherwise} \end{cases}$

For guards we have a predicate $\mathbf{e}(\cdot)$, that is defined in the obvious way for true (tt), conjunction and negation. For the other atomic predicates, we define $\mathbf{e}([F=G])$ to be true iff $\mathbf{e}_{\mathbf{c}}(F) = \mathbf{e}_{\mathbf{c}}(G) \neq \bot$, $\mathbf{e}([F:\mathcal{N}])$ iff $\mathbf{e}_{\mathbf{c}}(F) \in \mathcal{N}$ and $\mathbf{e}([F:\mathcal{M}])$ iff $\mathbf{e}_{\mathbf{c}}(F) \neq \bot$. Note that $\mathbf{e}([F:\mathcal{M}])$ iff $\mathbf{e}([F=F])$.

The set of actions $\mu \in \mathcal{A}$ is defined by $\mu ::= F(x) \mid (\nu \tilde{c}) \overline{F} G \mid \tau$, where \tilde{c} is a tuple of names. By abuse of notation, we write $\overline{F} G$ for $(\nu \tilde{c}) \overline{F} G$ when \tilde{c} is empty. We let $\operatorname{bn}(F(x)) := \{x\}$ and $\operatorname{bn}((\nu \tilde{c}) \overline{F} G) := \{\tilde{c}\}$. Moreover, we define the channel of a visible action as $\operatorname{ch}(F(x)) := F$ and $\operatorname{ch}((\nu \tilde{c}) \overline{F} G) := F$. The derivation rules for the concrete semantics are the C-rules displayed in Table 1, plus symmetric variants of (CSUM), (CPAR.) and (CCOM).

Symbolic semantics. The idea behind the symbolic semantics is to record, without checking, the necessary conditions for a transition as it is derived. Restrictions are still handled in the side conditions of the derivation rules, but all other constraints are simply collected in *transition constraints*. There are three major differences to other symbolic semantics [BD96, HL95], resulting from the presence of compound messages containing names.

- 1. We record the freshness of restricted names in the constraint separately, because of the complex guards and expressions.
- 2. We must use *abstract evaluation* to avoid unnecessary scope extrusion while deferring key correspondence checks.
- 3. The requirement that channels should be names and messages should be in \mathcal{M} need to be part of the transition constraint.
Table 1. Concrete and Symbolic Operational Semantics

We assume that the bound names of a process are pairwise different and different from its free names, and do not permit α -renaming during the derivation of a transition.

$$\begin{array}{l} \text{(Cout)} & \overline{\overline{G}\langle F\rangle.P} \xrightarrow{\overline{\mathbf{e}_{c}(G)} \mathbf{e}_{c}(F)} P & \text{if } \mathbf{e}([G:\mathcal{N}]) \text{ and } \mathbf{e}([F:\mathcal{M}]) \\ \\ \text{(CIN)} & \overline{G(x).P} \xrightarrow{\overline{\mathbf{e}_{c}(G)(x)}} P & \text{if } \mathbf{e}([G:\mathcal{N}]) & (CGUARD) \frac{P \xrightarrow{\mu} P'}{\phi P \xrightarrow{\mu} P'} & \text{if } \mathbf{e}(\phi) \\ \\ \text{(CCOM)} & \frac{P \xrightarrow{a(x)} P'}{P \mid Q \xrightarrow{\tau} (\nu \bar{v}) (\nu \bar{v}) \left(P' \begin{bmatrix} M_{/x} \end{bmatrix} \mid Q'\right)} & \text{if } \mathbf{e}([a=b]) \\ \\ \text{(CRES)} & \frac{P \xrightarrow{\mu} P'}{(\nu a) P \xrightarrow{\mu} (\nu a) P'} & \text{if } a \notin \mathbf{n}(\mu) & (CPAR) \frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q} \\ \\ \text{(COPEN)} & \frac{P \xrightarrow{(i\bar{v}) Z M} P'}{(\nu a) P \xrightarrow{(i\bar{v}) Z M} P'} & \text{if } c \neq a \in \mathbf{n}(M) & (CSUM) \frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'} \\ \\ \text{(SOUT)} & \overline{\overline{G}\langle F\rangle.P} & \overline{\overline{\mathbf{e}_{a}(G)} \mathbf{e}_{a}(F) \over (G:\mathcal{N}) \wedge [F:\mathcal{M}]} P & (SIN) & \overline{G(x).P} & \overline{\mathbf{e}_{a}(G)(x)}, P \\ \\ \text{(SOUT)} & \overline{\overline{Q}\langle F\rangle.P} & \frac{\overline{\mathbf{e}_{a}(G)}(\mathbf{e}_{a})}{(\nu \bar{v}) (i + P + Q)} P' & Q & (\nu \bar{v}) \overline{O^{T} F} \\ \\ \text{(SOUT)} & \overline{\overline{Q}\langle F\rangle.P} & \frac{\overline{\mathbf{e}_{a}(G)}(\mathbf{e}_{a})}{(\nu \bar{v}) (i + P + Q)} P' & Q & (\nu \bar{v}) \overline{O^{T} F} \\ \\ \\ \text{(SOUT)} & \overline{\overline{Q}\langle F\rangle.P} & \frac{\overline{\mathbf{e}_{a}(G)}(\mathbf{e}_{a})}{(\nu \bar{v}) (i + P + Q)} P' & Q & (\nu \bar{v}) \overline{O^{T} F} \\ \\ \\ \text{(SOUT)} & \overline{\overline{Q}\langle F\rangle.P} & \frac{\overline{\mathbf{e}_{a}(G)}(\mathbf{e}_{a})}{(\nu \bar{v}) (i + (p + Q + Q))} P' \\ \\ \\ \text{(SOUT)} & \overline{\overline{Q}\langle F\rangle.P} & \frac{\overline{\mathbf{e}_{a}(G)}(\mathbf{e}_{a})}{(\nu \bar{v}) (\phi + \phi^{+})} P' & Q & (\nu \bar{v}) \overline{\overline{O^{T} F}} \\ \\ \\ \text{(SOUT)} & \overline{\overline{Q}\langle F\rangle.P} & \frac{\overline{\mathbf{e}_{a}(G)}(\mathbf{e}_{a})}{(\nu \bar{v}) (\phi + \phi^{+})} P' \\ \\ \\ \\ \text{(SOUARD)} & \frac{P & \frac{\mu}{(\nu \bar{v})} \Phi'}{(\nu \bar{v}) (\phi + \phi^{+})} P' & Q & (\nu \bar{v}) \overline{\overline{O^{T} F}} \\ \\ \\ \text{(SGUARD)} & \frac{P & \frac{\mu}{(\nu \bar{v})} \Phi'}{(\nu \bar{v}) (\phi + \phi^{+})} P' \\ \\ \\ \\ \text{(SOPEN-MSG)} & \frac{P & \frac{(\nu \bar{v})}{(\mu \bar{v})} \Phi'}{(\nu \bar{v}) \phi} P' \\ \\ \\ \\ \text{(SOPEN-GRD)} & \frac{P & \frac{\mu}{(\nu \bar{v})} \Phi'}{(\nu \bar{v}) \Phi'} (\nu a) P' & \text{if } \mathbf{n}(\mu) \neq a \in \mathbf{n}(\phi) \\ \end{array} \right) \end{array}$$

TEAM LING

A symbolic transition is written $P \xrightarrow[(\nu \bar{c})\phi]{\mu} P'$, where $\mu \in \mathcal{A}$. In a transition constraint $(\nu \bar{c})\phi$ we have $\phi \in \mathcal{F}$ and \bar{c} is a tuple of names that are fresh in ϕ . As above, we omit $(\nu \bar{c})$ when \bar{c} is empty. The symbolic counterpart to concrete evaluation is *abstract evaluation* $\mathbf{e}_{\mathbf{a}}(\cdot) : \mathcal{E} \to \mathcal{E}$. Intuitively, it performs all decryptions in a term without checking that decryption and encryption keys correspond. Instead, when used in the derivation of a transition, we add this requirement to the transition constraint.

$$\hat{F} \stackrel{\mathbf{e}_{\mathbf{a}}}{\mapsto} \begin{cases} a & \text{if } \hat{F} = a \\ \mathsf{E}_{\mathbf{e}_{\mathbf{a}}(F)}\mathbf{e}_{\mathbf{a}}(G) & \text{if } \hat{F} = \mathsf{E}_{F}G \\ G' & \text{if } \hat{F} = \mathsf{D}_{F}G \text{ and } \mathbf{e}_{\mathbf{a}}(G) = \mathsf{E}_{F'}G' \\ \mathsf{D}_{\mathbf{e}_{\mathbf{a}}(F)}\mathbf{e}_{\mathbf{a}}(G) & \text{if } \hat{F} = \mathsf{D}_{F}G \text{ and } \not\exists F', G' \text{ such that } \mathbf{e}_{\mathbf{a}}(G) = \mathsf{E}_{F'}G' \\ \mathsf{op}(\mathbf{e}_{\mathbf{a}}(F)) & \text{if } \hat{F} = \mathsf{op}(F) \text{ and } \mathsf{op} \in \{\mathsf{H}, \mathsf{pub}, \mathsf{priv}\} \end{cases}$$

Symbolic transitions are defined as the smallest relation generated by the S-rules of Table 1 plus symmetric variants of (SSUM), (SPAR) and (SCOM). Compared to the concrete semantics, concrete evaluation is replaced by abstract evaluation in the rules (SOUT) and (SIN). When we encounter a guard, then the rule (SGUARD) simply adds it to the transition constraint. If a bound name occurs only in the transition constraint then, with (SOPEN-GRD), its scope is not extruded; it remains restricted in the resulting process, and also appears restricted in the transition constraint. Together with abstract evaluation, this rule prevents unnecessary scope extrusion, as seen in the following example. This is necessary to obtain the desired correspondence (Lemma 1).

Example 1. Let $P := (\nu b) \overline{a} \langle \mathsf{D}_b \mathsf{E}_b a \rangle . Q$ for some Q. Concretely, $P \xrightarrow{\overline{a} a} (\nu b) Q$. Symbolically we have that $P \xrightarrow[(\nu b) [a:\mathcal{N}] \land [\mathsf{D}_b \mathsf{E}_b a:\mathcal{M}]} (\nu b) Q$, where b is still bound. However, if the definition of (SOUT) did not include $\mathbf{e}_{\mathbf{a}}(\cdot)$, we would have $P \xrightarrow[(a:\mathcal{N}] \land [\mathsf{D}_b \mathsf{E}_b a]} Q$, where b is extruded.

Concrete transitions correspond to symbolic transitions with true constraints.

Lemma 1. $P \xrightarrow{\mu} P'$ iff $\exists \phi, \tilde{c}$ such that $P \xrightarrow{\mu} P'$ and $\mathbf{e}(\phi)$.

PROOF: By induction on the derivation of the transitions.

4 Bisimulations – Concrete and Symbolic

In the spi calculus, bisimulations must take into account the cryptographic knowledge of the observing environment—potentially a malicious attacker. To relate two processes P and Q, one usually seeks a bisimulation S such that $e \vdash P S Q$ for some environment e containing the free names of both processes.

168 J. Borgström et al.

In the following, we define two bisimulations and their respective notions of environment. Concrete bisimulation is a strong late version of hedged bisimulation as defined in [BN02]. Weak early hedged bisimulation is a variant of framed bisimulation [AG98] designed to be sound and complete with respect to barbed equivalence [BDP02]. Symbolic bisimulation is intended to enable automatic verification, while still being sufficiently complete with respect to the concrete bisimulation for the purpose of verifying security protocols (c.f. Section 6).

Concrete Bisimulation. The environment knowledge is stored in sets of pairs of messages, called *hedges*. The first message of a pair contributes to the knowledge about the first process; likewise the second message is related to the second process. Hedges evolved from the frame-theory pairs of [AG98] by dropping the frames. As a compact representation, we always work with *irreducible* hedges, where no more decryptions are possible. (Irreducibles are related to the notions of *core* in [BDP02] and *minimal closure seed* in [DSV03].) The set of message pairs that can be generated using the knowledge of the environment is called its *synthesis*. Since we want to use hedges also for the symbolic bisimulations, we do not *a priori* exclude pairs of non-message expressions in the hedges.

Definition 1 (Hedges). A hedge is a subset of $\mathcal{E} \times \mathcal{E}$. The synthesis $\mathcal{S}(h)$ of a hedge h is the smallest hedge containing h and satisfying

$$(\text{syn-enc}) \ \frac{(F_1, F_2) \in \mathcal{S}(h) \qquad (G_1, G_2) \in \mathcal{S}(h)}{(\mathsf{E}_{G_1} F_1, \mathsf{E}_{G_2} F_2) \in \mathcal{S}(h)}$$

$$(\text{SYN-OP}) \ \frac{(F_1,F_2) \in \mathcal{S}(h)}{(\mathsf{op}(F_1),\mathsf{op}(F_2)) \in \mathcal{S}(h)} \ \mathsf{op} \in \{ \,\mathsf{H},\mathsf{pub},\mathsf{priv} \,\}$$

The irreducibles $\mathcal{I}(\cdot)$ of a hedge are defined as

$$\begin{split} \mathcal{I}(h) &:= \mathcal{A}(h) \setminus \left(\{ (\mathsf{E}_{G_1}F_1, \mathsf{E}_{G_2}F_2) \mid (F_1, F_2), (G_1, G_2) \in \mathcal{S}(\mathcal{A}(h)) \} \\ & \cup \{ (\mathsf{op}(F_1), \mathsf{op}(F_2)) \mid (F_1, F_2) \in \mathcal{S}(\mathcal{A}(h)) \land \mathsf{op} \in \{\mathsf{H}, \mathsf{pub}, \mathsf{priv}\} \} \right) \end{split}$$

where the analysis $\mathcal{A}(h)$ is the smallest hedge containing h and satisfying

(ANA-DEC)
$$\frac{(\mathsf{E}_{G_1}F_1, \mathsf{E}_{G_2}F_2) \in \mathcal{A}(h) \qquad (G_1^{-1}, G_2^{-1}) \in \mathcal{S}(\mathcal{A}(h))}{(F_1, F_2) \in \mathcal{A}(h)}$$

We write $h \vdash F_1 \leftrightarrow F_2$ for $(F_1, F_2) \in \mathcal{S}(h)$. If h is a hedge, we let $h^{\mathrm{T}} := \{ (F_2, F_1) \mid (F_1, F_2) \in h \}, \pi_1(h) := \{ F_1 \mid (F_1, F_2) \in h \}$ and $\pi_2(h) := \{ F_2 \mid (F_1, F_2) \in h \}.$

A concrete environment $ce \in CE := 2^{M \times M}$, i.e., a hedge that only contains pairs of messages, is consistent if it is irreducible and the attacker cannot distinguish between the messages in $\pi_1(ce)$ and their counterparts in $\pi_2(ce)$. The attacker can (1) distinguish names from composite messages, (2) check message equality, (3) create public and private keys and hashes, and (4) encrypt and (5) decrypt messages with any key it can create. **Definition 2** (Concrete Consistency). A finite concrete environment ce is semi-consistent iff whenever $(M_1, M_2) \in ce$

- 1. If $M_1 \in \mathcal{N}$ then $M_2 \in \mathcal{N}$
- 2. If $(N_1, N_2) \in ce$ such that $M_1 = N_1$ then $M_2 = N_2$
- 3. If $M_1 = op(N_1)$ where $op \in \{H, pub, priv\}$ then $N_1 \notin \pi_1(\mathcal{S}(ce))$
- 4. If $M_1 = \mathsf{E}_{N_1} N'_1$ then $N_1 \notin \pi_1(\mathcal{S}(ce))$ or $N'_1 \notin \pi_1(\mathcal{S}(ce))$
- 5. If $M_1 = \mathbb{E}_{N_1} N'_1$ and $N_1^{-1} \in \pi_1(\mathcal{S}(ce))$ then $M_2 = \mathbb{E}_{N_2} N'_2$ such that $(N_1^{-1}, N_2^{-1}) \in \mathcal{S}(ce)$ and $(N'_1, N'_2) \in \mathcal{S}(ce)$. 6. If $(N_1, N_2) \in ce$ such that $M_1 = N_1^{-1}$ then $M_2 = N_2^{-1}$

ce is consistent iff both ce and ce^{T} are semi-consistent.

A concrete relation \mathcal{R} is a subset of $\mathbf{CE} \times \mathcal{P} \times \mathcal{P}$. \mathcal{R} is *consistent* if $ce \vdash P \mathcal{R} Q$ implies that *ce* is consistent. A concrete relation \mathcal{R} is symmetric if $ce \vdash P \mathcal{R} Q$ implies $ce^{\mathsf{T}} \vdash Q \mathcal{R} P$.

Intuitively, for two processes to be concretely bisimilar under a given concrete environment every *detected* transition of one of the processes must be simulated by a transition of the other process on a *corresponding* channel such that the updated environment is consistent.

Definition 3 (Concrete Bisimulation). A symmetric consistent concrete re*lation* \mathcal{R} *is a* concrete bisimulation if when $ce \vdash P \mathcal{R} Q$ and $P \xrightarrow{\mu_1} P'$ with

$$- \operatorname{bn}(\mu_1) \cap \operatorname{fn}(\pi_1(ce)) = \emptyset$$
 (bound names are fresh)

$$- \operatorname{ch}(\mu_1) \in \pi_1(ce) \text{ if } \mu_1 \neq \tau$$
 (the transition is detected)

then $Q \xrightarrow{\mu_2} Q'$ where

1. If $\mu_1 = \tau$ then $\mu_2 = \tau$ and $ce \vdash P' \mathcal{R} Q'$.

2. If $\mu_1 = a_1(x_1)$ then $\mu_2 = a_2(x_2)$ where $x_2 \notin fn(\pi_2(ce))$ and for all B, M_1, M_2 with $B \subset \mathcal{N} \times \mathcal{N}$ consistent and $-\pi_1(B)\setminus \operatorname{fn}(M_1)=\emptyset$ (all new names are needed) $-\pi_1(B) \cap (\operatorname{fn}(P) \cup \operatorname{fn}(\pi_1(ce))) = \emptyset = \pi_2(B) \cap (\operatorname{fn}(Q) \cup \operatorname{fn}(\pi_2(ce)))$ (new names are fresh)

 $\begin{array}{l} - ce \cup B \vdash M_1 \leftrightarrow M_2 \quad (M_1 \text{ and } M_2 \text{ are indistinguishable}) \\ we have \quad ce \cup B \cup \{(a_1, a_2)\} \vdash P' \begin{bmatrix} M_1 \\ m_1 \\ m_2 \end{bmatrix} \mathcal{R} \; Q' \begin{bmatrix} M_2 \\ m_2 \\ m_2 \end{bmatrix}. \end{array}$

3. If $\mu_1 = (\nu \tilde{c}_1) \overline{a_1} M_1$ then $\mu_2 = (\nu \tilde{c}_2) \overline{a_2} M_2$ where $\{\tilde{c}_2\} \cap \operatorname{fn}(\pi_2(ce)) = \emptyset$ and $\mathcal{I}(ce \cup \{(a_1, a_2), (M_1, M_2)\}) \vdash P' \mathcal{R} Q'$.

Concrete bisimilarity, written \sim_{c} , is the union of all concrete bisimulations.

In the definition above, we check channel correspondence by adding the channels to the environment. If they do not correspond, the resulting environment will not be consistent (Definition 2, item 2).

On process output we use $\mathcal{I}(\cdot)$ to construct the new environment after the transition. This entails applying all decryptions with keys that are known by the environment, producing the minimal extension of the environment *ce* with (M_1, M_2) . This extension may turn out to be inconsistent, signifying that the environment can distinguish corresponding messages from the two processes.

On process input any input that the environment can construct (i.e., satisfying $ce \cup B \vdash M_1 \leftrightarrow M_2$) must be considered. This is the main problem for automating bisimilarity checks, since the set of potential inputs is infinite. We now define a symbolic bisimulation for the spi-calculus, with the property that every simulated input action gives rise to only one new process pair.

Symbolic Bisimulation. As with concrete bisimulation, we need an environment to keep track of what an attacker has learned during a bisimulation game. As in the concrete case, a *symbolic environment* contains a hedge to hold the initial knowledge of an environment and the knowledge derived from messages received from the processes. Moreover, in a second hedge, we store the input variables that we come across when performing process inputs. Similarly to other symbolic bisimulations [HL95, BD96], we record the transition constraints accumulated by the processes. Finally, to know whether an input was performed before or after the environment learned a given message (e.g., the key of an encrypted message) the knowledge and the input variables are augmented with timing information.

Example 2. This example, inspired by [AG99], illustrates why we need to remember the order of received messages. Let $P := c(x).(\nu k) (\overline{c} \langle k \rangle. \overline{c} \langle \mathsf{D}_k x \rangle)$. Since the input of x happens *before* P publishes its private key k, x cannot be equal to a ciphertext encrypted with k. So, the output $\overline{c} \langle \mathsf{D}_k x \rangle$ can never execute.

Definition 4 (Symbolic Environments). A symbolic environment $se = (th, tw, (\phi_1, \phi_2))$ consists of the following three elements.

- 1. A timed hedge $th: \mathcal{E} \times \mathcal{E} \rightarrow \mathbb{N}$ representing the knowledge of the environment.
- 2. A timed variable set $tw : \mathcal{N} \times \dot{\mathcal{N}} \rightarrow \mathbb{N}$ containing earlier input variables.
- 3. A pair of formulae (ϕ_1, ϕ_2) that are the accumulated transition constraints.

The set of finite symbolic environments is denoted SE. We let $n_i(se) := fn(\pi_i(dom(th))) \cup \pi_i(dom(tw)) \cup fn(\phi_i)$ for $i \in \{1, 2\}$. To swap the sides of a timed hedge we define $th^T := \{(F_1, F_2) \mapsto th(F_2, F_1) \mid (F_2, F_1) \in dom(th)\}$. We take a snapshot of a timed hedge as $th|_t := \{(F, G) \mapsto th(F, G) \mid th(F, G) < t\}$.

Example 3. A symbolic environment related to Example 2 is se_2 where $se_n := (th_n, tw, (\phi_1, \phi_2))$ for $th_n := \{(k, k) \mapsto n, (\mathsf{D}_k x, \mathsf{D}_k x) \mapsto 3\}, tw := \{(x, x) \mapsto 1\}$ and $\phi_1 := \phi_2 := [\mathsf{D}_k x : \mathcal{M}].$

A symbolic environment can be understood as a concise description of a set of concrete environments, differing only in the instantiations of variables. Here, a *variable instantiation* is a pair of substitutions, that are applied to the knowledge of a symbolic environment. As in the concrete case, we may create some fresh names (*B* below) when instantiating variables. This definition of concretization does not constrain the substitutions or 'fresh' names, but see Definition 6.

Definition 5 (Concretization). Given $B \subset \mathcal{N} \times \mathcal{N}$ and substitutions σ_1, σ_2 we can concretize a timed hedge th into

 $\mathbf{C}^{B}_{\sigma_{1},\sigma_{2}}(th) := \mathcal{I}(\{(\mathbf{e}_{c}(F_{1}\sigma_{1}),\mathbf{e}_{c}(F_{2}\sigma_{2})) \mid (F_{1},F_{2}) \in \mathrm{dom}(th)\} \cup B).$ Note that $\mathbf{C}^{B}_{\sigma_{1},\sigma_{2}}(th) \in \mathbf{CE}$ if all evaluations are defined.

Example 4. We take $th_2 = \{(k, k) \mapsto 2, (D_k x, D_k x) \mapsto 3\}$ from Example 3.

If $\sigma_1 := \sigma_2 := \begin{bmatrix} \mathsf{E}_k a / x \end{bmatrix}$ then $\mathbf{C}_{\sigma_1, \sigma_2}^{\{(a,a)\}}(th_2) = \{(k, k), (a, a)\}.$ If $\rho_1 := \rho_2 := \begin{bmatrix} a / x \end{bmatrix}$ then $\mathbf{C}_{\rho_1, \rho_2}^{\{(a,a)\}}(th_2) = \mathcal{I}(\{(k, k), (\mathbf{e}_c(\mathsf{D}_k a), \mathbf{e}_c(\mathsf{D}_k a)), (a, a)\}),$ which is undefined since $\mathbf{e}_c(\mathsf{D}_k a) = \bot.$

A symbolic environment does not permit arbitrary variable instantiations. To begin with, the corresponding concretization must be defined. Furthermore, in order not to invalidate previous transitions that have taken place, we require the accumulated transition constraints to hold after variable instantiation. Finally, if a variable corresponds to an input performed at time t, then the message substituted for the variable must be synthesizable from the knowledge of the environment at that time, augmented with some fresh names B.

Definition 6 (se-Respecting Substitutions). A substitution pair (σ_1, σ_2) is called se-respecting with $B \subseteq \mathcal{N} \times \mathcal{N}$, written $se \vdash \sigma_1 \leftrightarrow_B \sigma_2$ iff

- 1. dom $(\sigma_i) = \pi_i(\text{dom}(tw))$ and $\mathbf{e}(\phi_i \sigma_i)$ for $i \in \{1, 2\}$.
- 2. If $(F_1, F_2) \in \text{dom}(th)$ then $\mathbf{e}_c(F_i \sigma_i)$ is defined for $i \in \{1, 2\}$.
- 3. If $(v_1, v_2) \in \operatorname{dom}(tw)$ then $\mathbf{C}^{B}_{\sigma_1, \sigma_2}(th|_{tw(v_1, v_2)}) \vdash \sigma_1(v_1) \leftrightarrow \sigma_2(v_2)$. 4. B is consistent (Definition 2) such that $\pi_i(B) \cap n_i(se) = \emptyset$ for $i \in \{1, 2\}$ and if $(b_1, b_2) \in B$ then $b_1 \in \operatorname{fn}(\operatorname{range}(\sigma_1))$ or $b_2 \in \operatorname{fn}(\operatorname{range}(\sigma_2))$.

Example 5. We take se_n as defined in Example 3 and let $\sigma_1 := \sigma_2 := [E_k a/x]$. If n = 0, then $se_n \vdash \sigma_1 \leftrightarrow_{\{(a,a)\}} \sigma_2$ since $\mathbf{C}^{\{(a,a)\}}_{\sigma_1,\sigma_2}(th_0|_{tw(x,x)}) = \mathcal{I}(\{(a,a), (k,k), (\mathbf{e}_c(\mathsf{D}_k\mathsf{E}_ka), \mathbf{e}_c(\mathsf{D}_k\mathsf{E}_ka))\}) = \{(a,a), (k,k)\}$ and $\{(a,a),(k,k)\} \vdash \mathsf{E}_k a \leftrightarrow \mathsf{E}_k a.$

If n > 1 (k becomes known strictly after x was input) then we do not have $se_n \vdash \sigma_1 \leftrightarrow_B \sigma_2$ for any B since we cannot synthesize $\mathsf{E}_k a$ before knowing k.

In contrast to the concrete case, there are two different ways for a symbolic environment to be inconsistent. (1) If one of the concretizations of the environment is inconsistent: The attacker can distinguish between the messages received from the two processes. (2) If there is a concretization such that, after substituting, one of the accumulated transition constraints holds but the other does not: One of the processes made a transition that was not simulated by the other.

Definition 7 (Symbolic Consistency). Let $se = (th, tw, (\phi_1, \phi_2)) \in SE$ be a symbolic environment. se is consistent if for all B, σ_1, σ_2 we have that

- 1. $se \vdash \sigma_1 \leftrightarrow_B \sigma_2$ implies that $\mathbf{C}^B_{\sigma_1,\sigma_2}(th)$ is consistent;
- 2. $(th, tw, (tt, tt)) \vdash \sigma_1 \leftrightarrow_B \sigma_2 \text{ and } \pi_i(B) \cap \operatorname{fn}(\phi_i) = \emptyset \text{ for } i \in \{1, 2\}$ implies that $\mathbf{e}(\phi_1 \sigma_1)$ iff $\mathbf{e}(\phi_2 \sigma_2)$.

172 J. Borgström et al.

The definition of symbolic bisimilarity is similar to the concrete case. To see if a transition needs to be simulated, we search a concretization under which the transition takes place concretely and is detected. On input, we simply add the input variables to the timed variable set. For all transitions, we add the constraints to the environment. The consistency of the updated environment implies that the simulating transition is detected, and that the channels correspond.

A symbolic relation \mathcal{R} is a subset of $\mathbf{SE} \times \mathcal{P} \times \mathcal{P}$. \mathcal{R} is symmetric if $se \vdash P \mathcal{R} Q$ implies that $(th^{\mathrm{T}}, tw^{\mathrm{T}}, (\phi_2, \phi_1)) \vdash Q \mathcal{R} P$. \mathcal{R} is consistent if se is consistent whenever $se \vdash P \mathcal{R} Q$.

Definition 8 (Symbolic Bisimulation). A symmetric consistent symbolic relation
$$\mathcal{R}$$
 is a symbolic bisimulation if
whenever $(\underline{th}, tw, (\phi_1, \phi_2)) \vdash P \mathcal{R} Q$ and $P \xrightarrow{\mu_1}_{(\nu \tilde{d}_1)\psi_1} P'$ such that
 $-(\{\tilde{d}_1\} \cup \operatorname{bn}(\mu_1)) \cap \operatorname{n}_1(th, tw, (\phi_1, \phi_2)) = \emptyset$ (bound names are fresh)
 $-$ there exist σ_1, σ_2, B with $se \vdash \sigma_1 \leftrightarrow_B \sigma_2$ and
 $\bullet e(\psi_1 \sigma_1)$ (possible)
 $\bullet \operatorname{ch}(\mu_1) \in \pi_1(\mathbf{C}^B_{\sigma_1,\sigma_2}(th))$ if $\mu_1 \neq \tau$ (detectable)
 $\bullet \pi_1(B) \cap (\{\tilde{d}_1\} \cup \operatorname{bn}(\mu_1) \cup \operatorname{fn}(P)) = \emptyset$ (created names are fresh)
then $Q \xrightarrow{\mu_2}_{(\nu \tilde{d}_2)\psi_2} Q'$ with $T := \max(\operatorname{range}(th) \cup \operatorname{range}(tw))$ where
1. If $\mu_1 = \tau$ then $\mu_2 = \tau, \{\tilde{d}_2\} \cap \operatorname{n}_2(se) = \emptyset$ and
 $(th, tw, (\phi_1 \land \psi_1, \phi_2 \land \psi_2)) \vdash P' \mathcal{R} Q'.$
2. If $\mu_1 = F_1(x_1)$ then $\mu_2 = F_2(x_2), \{\tilde{d}_2 x_2\} \cap \operatorname{n}_2(se) = \emptyset$ and
 $(th \cup th', tw \cup \{(x_1, x_2) \mapsto T+1\}, (\phi_1 \land \psi_1, \phi_2 \land \psi_2)) \vdash P' \mathcal{R} Q'$ where
 $th' := \{(F_1, F_2) \mapsto t \mid t := th(F_1, F_2) \text{ if defined, else } t := T+1\}.$
3. If $\mu_1 = (\nu \tilde{c}_1) F_1 G_1$ then $\mu_2 = (\nu \tilde{c}_2) F_2 G_2, \{\tilde{d}_2 \tilde{c}_2\} \cap \operatorname{n}_2(se) = \emptyset$ and
 $(th \cup th', tw, (\phi_1 \land \psi_1, \phi_2 \land \psi_2)) \vdash P' \mathcal{R} Q'$ where

$$th' := \{ \mathbf{F}' \mapsto T+1 \mid \mathbf{F}' \in \mathcal{I}(\operatorname{dom}(th) \cup \{(F_1, F_2), (G_1, G_2)\}) \setminus \operatorname{dom}(th) \}.$$

Symbolic bisimilarity, written \sim_s , is the union of all symbolic bisimulations.

Theorem 1. Whenever $se \vdash P \sim_s Q$ and $se \vdash \sigma_1 \leftrightarrow_B \sigma_2$ with $\operatorname{fn}(P) \cap \pi_1(B) = \emptyset = \operatorname{fn}(Q) \cap \pi_2(B)$, we have that $\mathbf{C}^B_{\sigma_1,\sigma_2}(th) \vdash P\sigma_1 \sim_c Q\sigma_2$.

PROOF: To prove this theorem, we must verify two things.

- 1. Any concrete transition of $P\sigma_1$ that must be simulated by $Q\sigma_2$ under the concrete environment $\mathbf{C}^B_{\sigma_1,\sigma_2}(th)$ has a corresponding symbolic transition of P that must be simulated by Q under *se*.
- 2. If a symbolic transition of P is simulated by Q under *se*, and has a corresponding concrete transition of $P\sigma_1$ that must be simulated by $Q\sigma_2$ under $C^B_{\sigma_1,\sigma_2}(th)$, then $Q\sigma_2$ can simulate the concrete transition. Moreover, the process pairs and environments after the transition are related by a suitable extension of (σ_1, σ_2) .

By this theorem, symbolic bisimilarity is a sound approximation to concrete bisimilarity and, by transitivity, barbed equivalence. A weak version of symbolic bisimulation may be defined in the standard fashion.

5 Example

We prove that the equation of the example in §1 holds.

We start with a symbolic environment in which the message m is a variable: We let $th := \{(a, a) \mapsto 0, (f, f) \mapsto 0\}$, $tw := \{(m, m) \mapsto 1\}$ and se := (th, tw, (tt, tt)). Note that we give m a later time than a and f, in order to permit occurrences of a and f in the message.

Proposition 1. $se \vdash (\nu \underline{k}) (\underline{A} | \underline{B}) \sim_{s} (\nu k) (A | B)$

PROOF: We let $g_{F(x)} := [F:\mathcal{N}], g^{\overline{F}G} := [F:\mathcal{N}] \wedge [G:\mathcal{M}]$ and $g_{F(x)}^{\overline{F'}G} := g_{F(x)} \wedge g^{\overline{F'}G} \wedge [F=F']$. We write $pwd(\tilde{x})$ to denote that \tilde{x} is a tuple of *pair-wise* different names. The symmetric closure of the following set is a symbolic bisimulation. $\{((th, tw, tt, tt), (\nu \underline{k}) (\underline{A} | \underline{B}), (\nu k) (A | B)), ((th, tw, (g_{a(y)}^{\overline{a} \in \underline{k}m}, g_{a(x)}^{\overline{a} \in \underline{k}m})), (\nu \underline{k}) (\mathbf{0} | [D_{\underline{k}} \underline{E}_{\underline{k}}m : \mathcal{M}] \overline{f} \langle m \rangle), (\nu k) (\mathbf{0} | \overline{f} \langle D_{k} \underline{E}_{k}m \rangle)), ((th \cup \{(m, m) \mapsto 2\}, tw, (g_{a(y)}^{\overline{a} \in \underline{k}m} \wedge g^{\overline{f}m} \wedge [D_{\underline{k'}} \underline{E}_{\underline{k'}}m : \mathcal{M}], g_{a(x)}^{\overline{a} \in \underline{k}m} \wedge g^{\overline{f} D_{k'} \underline{E}_{k'}m})),$

 $\begin{array}{l} (\nu\underline{k}) (\mathbf{0} \mid \mathbf{0}), (\nu k) (\mathbf{0} \mid \mathbf{0})), \\ ((th, tw \cup \{(y, x) \mapsto 2\}, (g_{a(y)}, g_{a(x)})), (\nu\underline{k}) (\underline{A} \mid [\mathsf{D}_{\underline{k}}y : \mathcal{M}]\overline{f}\langle m \rangle), (\nu k) (A \mid \overline{f}\langle \mathsf{D}_{k}x \rangle)), \\ ((th \cup \{(\underline{E}_{\underline{k}}m, \underline{E}_{k}m) \mapsto 3\}, tw \cup \{(y, x) \mapsto 2\}, (g_{a(y)} \land g^{\overline{\alpha}} \underline{E}_{\underline{k}}^{m}, g_{a(x)} \land g^{\overline{\alpha}} \underline{E}_{k}m)), \\ (\mathbf{0} \mid [\underline{D}_{\underline{k}}y : \mathcal{M}]\overline{f}\langle m \rangle), (\mathbf{0} \mid \overline{f}\langle \mathsf{D}_{k}x \rangle)), \\ ((th \cup \{(\underline{E}_{\underline{k}}m, \underline{E}_{k}m) \mapsto 2\}, tw, (g^{\overline{\alpha}} \underline{E}_{\underline{k}}m, g^{\overline{\alpha}} \underline{E}_{k}m), (\mathbf{0} \mid \underline{B}), (\mathbf{0} \mid \underline{B})), \\ ((th \cup \{(\underline{E}_{\underline{k}}m, \underline{E}_{k}m) \mapsto 2\}, tw \cup \{(y, x) \mapsto 3\}, (g^{\overline{\alpha}} \underline{E}_{\underline{k}}m \land g_{a(y)}, g^{\overline{\alpha}} \underline{E}_{k}m \land g_{a(x)})), \\ (\mathbf{0} \mid [\underline{D}_{\underline{k}}y : \mathcal{M}]\overline{f}\langle m \rangle), (\mathbf{0} \mid \overline{f}\langle \mathsf{D}_{k}x \rangle)), \\ ((th \cup \{(\underline{E}_{\underline{k}}m, \underline{E}_{k}m) \mapsto 2, (m, \underline{D}_{k}x) \mapsto 4\}, tw \cup \{(y, x) \mapsto 3\}, \\ (g^{\overline{\alpha}} \underline{E}_{\underline{k}}m \land g_{a(y)} \land g^{\overline{f}}m \land [\underline{D}_{\underline{k}}y : \mathcal{M}], g^{\overline{\alpha}} \underline{E}_{\underline{k}}m \land g_{a(x)}) \land g^{\overline{f}} \underline{D}_{\underline{k}}x), (\mathbf{0} \mid \mathbf{0}), (\mathbf{0} \mid \mathbf{0})) \\ \mid \mathrm{pwd}(a, f, m, y, k, k') \text{ and } \mathrm{pwd}(a, f, m, x, k, k')\} \end{array}$

Note that the set itself is infinite, but that this infinity only arises from the possible different choices of bound names. Effectively, the bisimulation contains only $7 \cdot 2 = 14$ process pairs. We only check the element

$$(\underbrace{(th,tw\cup\{(y,x)\mapsto 2\},(g_{a(y)},g_{a(x)}))}_{se'},(\nu\underline{k})(\underline{A}\,|\,[\mathsf{D}_{\underline{k}}y:\mathcal{M}]\overline{f}\langle m\rangle),(\nu k)\,(A\,|\,\overline{f}\langle\mathsf{D}_{k}x\rangle)).$$

Consistency. If $se' \vdash \sigma_1 \leftrightarrow_B \sigma_2$ then $\mathbf{C}^B_{\sigma_1,\sigma_2}(th) = B \cup \{(a,a), (f,f)\}$, which is consistent by the consistency of *B* since $\{a, f\} \cap (\pi_1(B) \cup \pi_2(B)) = \emptyset$. We also have $\mathbf{e}(g_{a(y)}\sigma_1) = \mathbf{e}([a:\mathcal{N}])$ which is true independently of σ_1 , and $\mathbf{e}(g_{a(x)}\sigma_2) = \mathbf{e}([a:\mathcal{N}])$ which is also always true. Thus se' is consistent.

Transition 1. $(\nu \underline{k}) (\underline{A} | [D_{\underline{k}} y : \mathcal{M}] \overline{f} \langle m \rangle) \xrightarrow{(\nu \underline{k}) \overline{a} \mathsf{E}_{\underline{k}} m}{g^{\overline{a} \mathsf{E}_{\underline{k}} m}} \mathbf{0} | [D_{\underline{k}} y : \mathcal{M}] \overline{f} \langle m \rangle$ has to be simulated, since if we let $\rho_1 := \rho_2 := [a'_m] [a'_f]$ then we have that $se' \vdash \rho_1 \leftrightarrow_{\emptyset} \rho_2$ and $a \in \{a\} = \pi_1(\mathbf{C}^{\emptyset}_{\rho_1,\rho_2}(th))$. We simulate it by $(\nu k) (A | \overline{f} \langle D_k x \rangle) \xrightarrow{(\nu k) \overline{a} \mathsf{E}_k m}{g^{\overline{a} \mathsf{E}_k m}} \mathbf{0} | \overline{f} \langle D_k x \rangle.$

TEAM LING

Transition 2. First we α -rename to avoid clashes with environment names. $(\nu \underline{k'}) (\underline{A} | [D_{\underline{k'}}y:\mathcal{M}]\overline{f}\langle m \rangle) \xrightarrow{\overline{f}m} (\nu \underline{k'}) g^{\overline{f}m \wedge [D_{\underline{k'}}y:\mathcal{M}]} (\nu \underline{k}) (\underline{A} | \mathbf{0})$ does not need to be simulated: $\mathbf{e}([\mathsf{D}_{k'}\sigma(y):\mathcal{M}])$ holds iff $\sigma(y) = \mathsf{E}_{k'}M$ for some M, but $\underline{k'}$ cannot be in $n(range(\sigma))$ since it is bound in the transition constraint.

6 Sources of Incompleteness

The following examples show sources of incompleteness of the proposed "very late" symbolic bisimulation. All these examples start from the same symbolic environment $se := (\{(a, a) \mapsto 0\}, \emptyset, (tt, tt))$. Since se has no variables, it has only one concretization $ce := \mathbf{C}^{\emptyset}_{\epsilon,\epsilon}(\{(a,a) \mapsto 0\}) = \{(a,a)\}.$

In general, symbolic bisimulations let us postpone the "instantiation" of input variables until the moment they are actually used, leading to a stronger relation. In the pi calculus this was addressed using ϕ -decompositions [BD96]. We let

$$P_1 := a(x).(\overline{a}\langle a
angle + [x=a]\overline{a}\langle a
angle.\overline{a}\langle a
angle)
onumber \ Q_1 := a(x).(\overline{a}\langle a
angle + \overline{a}\langle a
angle.[x=a]\overline{a}\langle a
angle).$$

Proposition 2. $ce \vdash P_1 \sim_c Q_1$ but $se \vdash P_1 \not\sim_s Q_1$.

The next example shows that the requirement that the collected transition guards should be indistinguishable gives rise to some incompleteness, that we conjecture could be removed by allowing decompositions of the guards. We let

$$\begin{split} P_2 &:= a(x).\overline{a}\langle a \rangle \\ Q_2 &:= a(x).([x=a]\overline{a}\langle a \rangle \,|\, \neg [x=a]\overline{a}\langle a \rangle). \end{split}$$

Proposition 3. $ce \vdash P_2 \sim_c Q_2$ but $se \vdash P_2 \not\sim_s Q_2$.

PROOF: Since an output action of Q_2 always has an extra equality or disequality constraint compared to the output action of P_2 , the resulting symbolic environment is not consistent. In contrast, concrete bisimulation instantiates the input at once, killing one of the output branches of Q_2 .

Incompleteness also arises from the fact that we choose not to calculate the precise conditions for the environment to detect a process action. We let

$$\begin{split} P_{3} &:= a(x).(\nu k) \,\overline{a} \langle \mathsf{E}_{k} x \rangle.(\nu m) \,\overline{a} \langle \mathsf{E}_{\mathsf{E}_{k} a} m \rangle.P_{3}' \qquad P_{3}' := \overline{m} \langle a \rangle \\ Q_{3} &:= a(x).(\nu k) \,\overline{a} \langle \mathsf{E}_{k} x \rangle.(\nu m) \,\overline{a} \langle \mathsf{E}_{\mathsf{E}_{k} a} m \rangle.Q_{3}' \qquad Q_{3}' := [x = a] \overline{m} \langle a \rangle. \end{split}$$

Proposition 4. $ce \vdash P_3 \sim_c Q_3$ but $se \vdash P_3 \not\sim_s Q_3$.

PROOF: The output action of P'_3 is detected iff the first input was equal to *a*: Then the first message is the key of the second message. Since this constraint is not added to the symbolic environment but the explicit equality constraint of Q'_3 is, we have an inconsistent symbolic environment after the final outputs.

Impact. We have seen above that processes that are barbed equivalent but differ in the placement of guards may not be symbolically bisimilar. However, we contend that this incompleteness will not affect the verification of secrecy and authenticity properties of security protocols. For secrecy, we want to check whether two instances of the protocol with different messages (or symbolic variables) are bisimilar, so there is no change in the structure of the guards. For authenticity, we conjecture that the addition of guards in the specification only triggers the incompleteness if they relate to the observability of process actions (c.f. Proposition 4), something that should never occur in real-world protocols.

7 Conclusions

Contribution. We have given a general symbolic operational semantics for the spi calculus, including the rich guard language of [BDP02] and allowing complex keys and public-key cryptography. We also propose the, to our knowledge, first symbolic notion of bisimilarity for the spi calculus, and prove it a sound approximation of concrete hedged bisimilarity.

Mechanizing Equivalence Checks. Ultimately, we seek mechanizable (efficiently computable) ways to perform equivalence checks. Hüttel [Hüt02] showed decidability of bisimilarity checking by giving a "brute-force" decision algorithm for framed bisimulation in a language of only finite processes. However, this algorithm is not practically implementable, generating $\gg 2^{2^{20}}$ branches for each input of the Wide-mouthed Frog protocol of [AG99].

Ongoing and Future Work We are currently working on an implementation of this symbolic bisimilarity with a guard language not including negation; the crucial point is the infinite quantifications in the definition of environment consistency. As in [Bor01], it turns out to be sufficient to check a finite subset of the environment-respecting substitution pairs: the minimal elements of a refinement preorder. However, the presence of consistency makes for a significant difference in the refinement relation.

Moreover, the symbolic bisimilarity presented in this paper is a compromise between the complexity of its definition and the degree of completeness; we have refined proposals that we conjecture will provide full completeness. We also conjecture that a slightly simplified version of our symbolic bisimulation could be used for the applied pi-calculus [AF01]. In this setting, any mechanization would depend heavily on the chosen message language and equivalence.

References

[AF01] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. of POPL '01*, pages 104–115, 2001.

- [AG98] M. Abadi and A. D. Gordon. A bisimulation method for cryptographic protocols. *Nordic Journal of Computing*, 5(4):267–303, 1998.
- [AG99] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, 1999.
- [AL00] R. M. Amadio and D. Lugiez. On the Reachability Problem in Cryptographic Protocols. In *Proc. of CONCUR 2000*, pages 380–394, 2000.
- [BD96] M. Boreale and R. De Nicola. A symbolic semantics for the π -calculus. Information and Computation, 126(1):34–52, 1996.
- [BDP02] M. Boreale, R. De Nicola, and R. Pugliese. Proof techniques for cryptographic processes. SIAM Journal on Computing, 31(3):947–986, 2002.
- [BN02] J. Borgström and U. Nestmann. On bisimulations for the spi calculus. In Proc. of AMAST 2002, pages 287–303, 2002. Full version: EPFL Report IC/2003/34. Accepted for Mathematical Structures in Computer Science.
- [Bor01] M. Boreale. Symbolic Trace Analysis of Cryptographic Protocols. In *Proc.* of *ICALP 2001*, pages 667–681, 2001.
- [Cor03] V. Cortier. Vérification automatique des protocoles cryptographiques. PhD thesis, École Normale Supérieure de Cachan, 2003.
- [CS02] H. Comon and V. Shmatikov. Is it possible to decide whether a cryptographic protocol is secure or not? *Journal of Telecommunications and Information Technology*, 4:5–15, 2002.
- [DSV03] L. Durante, R. Sisto, and A. Valenzano. Automatic testing equivalence verification of spi-calculus specifications. ACM Transactions on Software Engineering and Methodology, 12(2):222–284, Apr. 2003.
- [FA01] M. Fiore and M. Abadi. Computing Symbolic Models for Verifying Cryptographic Protocols. In 14th IEEE Computer Security Foundations Workshop, pages 160–173, 2001.
- [HL95] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Comput. Sci.*, 138(2):353–389, 1995.
- [Hui99] A. Huima. Efficient Infinite-State Analysis of Security Protocols. In *FLOC* Workshop on Formal Methods and Security Protocols, 1999.
- [Hüt02] H. Hüttel. Deciding framed bisimilarity. In Proc. of INFINITY, 2002.
- [San96] D. Sangiorgi. A theory of bisimulation for the π -calculus. Acta Informatica, 33:69–97, 1996.
- [VM94] B. Victor and F. Moller. The Mobility Workbench a tool for the π -calculus. In *Proc. of CAV '94*, pages 428–440, 1994.

A Symbolic Decision Procedure for Cryptographic Protocols with Time Stamps* (Extended Abstract)

Liana Bozga, Cristian Ene, and Yassine Lakhnech

VERIMAG, 2 av. de Vignate, 38610 Grenoble, France {Liana.Bozga, Cristian.Ene, Yassine.Lakhnech}@imag.fr

Abstract. We present a symbolic decision procedure for time-sensitive cryptographic protocols with time-stamps. Our decision procedure deals with secrecy, authentication and any property that can be described as an invariance property.

1 Introduction

Cryptographic protocols are mandatory to ensure secure transactions in an open environment. They must be able to guarantee confidentiality, authentication and other security properties despite the fact that transactions take place in face of an intruder who may have complete control of a network, i.e. who may monitor, delete, alter or redirect messages. To achieve this goal these protocols rely upon cryptographic primitives and fresh nonces. The cryptographic primitives allow to encrypt messages with keys such that only a principal that owns the inverse key is able to extract the plain text from the cipher text; while nonces are used to prevent from replaying and redirecting messages. Nonces are usually implemented as randomly generated numbers. Now, such an implementation is not always feasible, and therefore, some cryptographic protocols rely upon timestamps or counters instead of nonces. Timestamps are then used by recipients to verify timeliness of the message and recognize and reject replays of messages communicated in the past. The problem is, however, that while the value of a nonce is not predictable, the value of a counter or a timestamps is. Hence, replacing nonces by counters or timestamps can produce new attacks. Moreover, a verification method has to take into account this predictability feature.

Most of the automatic verification methods for cryptographic protocols consider time-independent protocols [17,16,15,9] with the exception of [8,13].

In this paper, we present a model for time-dependent cryptographic protocols and a corresponding decidability result for the verification of a large class of properties. Our decidability holds for the Dolev-Yao model, i.e. assuming an active intruder, extended with rules associated to timestamps. Although, the

^{*} This work has been partially suppoted by the projects ACI-SI ROSSIGNOL http://www.cmi.univ-mrs.fr/~lugiez/aci-rossignol.html and PROUVE-03V360.

decidability we present applies to bounded protocols, i.e., when a fixed number of sessions are considered, our model clearly identifies the main ingredients to be included in a general model. It is useful to notice that the verification problem is undecidable for unbounded sessions.

Besides general models for distributed systems that can be used to model security protocols under some restrictions such as Timed *CSP* and *MSR* (multiset rewriting over first-order atomic formulae), there are very few models for timed protocols [11].

Contributions: The first contribution of this paper is a general model for timed cryptographic protocols. We include in our model clocks, time variables and timestamps. Clocks are variables that range over the time domain and advance with the same rate as time. Each agent has its own set clocks that he can reset. That is clocks can be used to measure the time that elapses between two events, for instance, sending a message and receiving the corresponding response. Also, we allow a global clock that is never reset and that can be read and tested by all participants. Time variables correspond to timestamps in received messages. Such values can be stored and used together with clocks to put conditions on the acceptance of a message.

A second contribution of this paper is the decidability of the verification of a large class of security properties including secrecy and authentication. We consider a rich class of reachability properties that allow to specify secrecy, temporary secrecy and authentication. In fact, we introduce a logic that allows to describe secrecy, temporary secrecy, equalities between terms and control points. Then, given a bounded protocol Π and two formulae in this logic Φ and Ψ , the reachability problem we consider is whether there is a run of Π that starts in a configuration that satisfies Φ and reaches a configuration that satisfies Ψ .

We device a symbolic algorithm that given a property described by a formula Ψ in this logic and given a bounded protocol computes the set of configurations that reaches Ψ . This algorithm uses symbolic constraints (logic formulae) to describe sets of configurations. The logic we introduce combines constraints on the knowledge of the intruder with time constraints on clock values and time variables. To show effectiveness of our verification method we show:

- 1. that for each action of our model we can express the predecessor configurations of a set of configurations as a formula. We consider input, output and time actions.
- 2. Then, we show decidability of the satisfiability problem for our logic.

It should be clear that even in the case of bounded protocols the underlying transition system is infinite state even if we do not consider timing aspects, This is because the size of messages is unbounded and the intruder is modeled as an infinitely iterating process. Handling time constraints and unbounded messages symbolically and automatically is the distinguishing feature of our verification method. Most of the work on timed cryptographic protocols uses theorem-provers or finite-state model-checking [1,4,8,13,14]. While the first needs human help, the second relies on typing assumptions and assumption on the time window to bound the search space. In [8], the authors make a semiautomated analysis on a Timed *CSP* model of Wide Mouth Frog protocol, and use *PVS* to discharge proof obligations in order to find an invariant property. In [14], a timed-authentication attack on Wide Mouth Frog protocol is found, using a model with discrete time and with an upper bound on the time window. Differently from [8,14], our method can be used for **automatic** verification of timed cryptographic protocols **without imposing any restrictions on the time model** (i.e. we can handle continuous time, and we need no upper bounds on the time window). Closest of our work is [6] which presents a verification method for timed protocols considering unbounded sessions. This paper does not, however, present a decidability result.

2 The Protocol and Intruder Model

A model for cryptographic protocols fixes on one hand the intruder capabilities and on the other the actions the principals can perform. In this section, we extend our model for cryptographic protocols [2] with timestamps. The untimed aspects of our model are fairly standard; it is the so-called Dolev-Yao model. But first we have to define the messages that can be sent.

Preliminaries. Let \mathcal{X} be a countable set of variables and let \mathcal{F}^i be a countable set of function symbols of arity *i*, for every $i \in \mathbb{N}$. Let $\mathcal{F} = \bigcup_{i \in \mathbb{N}} \mathcal{F}^i$. The set of *terms over* \mathcal{X} and \mathcal{F} , is denoted by $\mathcal{T}(\mathcal{X}, \mathcal{F})$. Ground terms are terms with no variables. We denote by $\mathcal{T}(\mathcal{F})$ the set of ground terms over \mathcal{F} . For any $t_1, t_2 \in \mathcal{T}(\mathcal{X}, \mathcal{F})$, we denote with $\mu(t_1, t_2)$ the most general unifier (mgu) of t_1 and t_2 , if it exists. We denote by $\Gamma(\tilde{x})$ the set consisting of ground substitutions with domain \tilde{x} . Given a tree t, we write t(p) to denote the symbol at position pin t and $t_{|p}$ to denote the subterm of t at position p.

If $w_1, w_2 \in \Sigma^*$ are words over an alphabet Σ , then we denote by $w_2^{-1}w_1$ the word obtained from w_1 after removing the prefix w_2 .

Messages and Terms. We fix the time domain to be the set of non-negative real numbers. Our results hold also when we consider the natural numbers instead.

Let \mathcal{X} denote the set of variables that range over terms. Let \mathcal{A} be a set of constant symbols with $\mathbb{R}_{\geq 0} \subset \mathcal{A}$ and $\mathcal{F} = \mathcal{A} \cup \{\text{encr, pair}\}$. We consider terms build from constant symbols in \mathcal{A} , clocks in \mathcal{C} and time variables in \mathcal{Y} using the function symbols in \mathcal{F} . As usual, we write (m_1, m_2) for $\text{pair}(m_1, m_2)$ and $\{m\}_k$ instead of encr(m, k). A *Clock-free term* is a term in which no clock appears; time variables and timestamps may appear in a clock-free term. We denote the set of clock-free terms by $\mathcal{T}(\mathcal{X} \cup \mathcal{Y}, \mathcal{F})$. *Messages* are ground (variable-free) terms in $\mathcal{T}(\mathcal{X} \cup \mathcal{Y}, \mathcal{F})$; we denote by $\mathcal{M} = \mathcal{T}(\mathcal{F})$ the set of messages. For conciseness, we write \mathcal{T} instead of $\mathcal{T}(\mathcal{X} \cup \mathcal{Y}, \mathcal{F})$ and \mathcal{T}_c instead of $\mathcal{T}(\mathcal{X} \cup \mathcal{Y} \cup \mathcal{C}, \mathcal{F})$.

In addition to the usual terms considered in Dolev-Yao model, we add:

1. Clocks, i.e. variables that range over the underlying time model. We denote the set of clocks by C.

- 2. Timestamps, that is values in the time domain.
- 3. Time variables, that is variables that range over the time domain. We denote by \mathcal{Y} the set time variables.

It is important to understand the difference between these three disjoint sets of variables: a timestamp is just a constant; clocks and time variables are variables. The difference is that the value of a clock advances with rate one with time while the value of a time variable does not. A time variable is simply a variable that ranges over the time domain.

The Intruder's Message Derivation Capabilities. We use the usual model of Dolev and Yao [7] augmented with the axiom: If $r \in \mathbb{R}_{>0}$ then $E \vdash r$. The axiom represents the fact that the intruder can guess every possible time-stamp, i.e. time value. As usual, we write $E \vdash m$, when m is derivable from E using the augmented Dolev-Yao model. For a term t, we use the notation $E \vdash t$ to denote that there exists a substitution $\sigma: \mathcal{X} \to \mathcal{M}$ such that $E \vdash t\sigma$. Given a term t, a position p in t is called *non-critical*, if it is not a key position; otherwise it is called *critical*.

2.1 Process Model

Timed cryptographic protocols are build from timed actions. Here, we consider two types of actions: message input and message output. A time constraint is associated to an action and describes when the action is possible.

Definition 1 (Time Constraints). Time constraints are boolean combinations of linear constraints on clocks and time variables and they are defined by:

$$g ::= \top \mid \sum_{i=1}^{n} a_i c_i + \sum_{j=1}^{m} b_j T_j \bowtie d \mid g_1 \land g_2 \mid g_1 \lor g_2$$

where $m, n \in \mathbb{N}$, $c_i \in \mathcal{C}$ are clocks, $T_i \in \mathcal{Y}$ are time variables, $a_i, b_i \in \mathbb{Z}$, $d \in \mathbb{Z}$, and $\bowtie \in \{<,\leq\}$. The set of time constraints is denoted by TC.

A time constraint is interpreted with respect to a valuation ν defined over a finite set of clocks $\{c_1, \ldots, c_n\}$ that associates values in the time domain to clocks, and a substitution σ that assigns ground clock-free terms to variables, thus in particular, values to the time variables. The interpretation of a time constraint, denoted by $[g]_{\nu,\sigma}$, is defined as usual. Then (ν,σ) is said to be a *model* for a time constraint g, if $[g]_{\nu,\sigma} = 1$.

Given a time constraint g and a set \mathcal{R} of clocks, we denote by $g[\mathcal{R}]$ the time constraint obtained by substituting 0 for all clocks in \mathcal{R} . We also use the notation q + d to denote the time constraint obtained from q by substituting each clock c in g by c + d.

Definition 2 (Actions and Protocols). We consider input and output actions:

- An input action is of the form $l \xrightarrow{g,\mathcal{R},?t(\bar{x})} l'$, where $g \in \mathcal{TC}$ is a time constraint called the guard,

 - $t(\tilde{x}) \in \mathcal{T}$ is a term and $\tilde{x} \subseteq \mathcal{X} \cup \mathcal{Y}$ is the set of variables instantiated by the input action.

- $\mathcal{R} \subseteq \mathcal{C}$ is a subset of clocks
- l, l' are labels
- An output action is of the form $l \xrightarrow{g,\mathcal{R},lt_c} l'$ where g, l,l' and \mathcal{R} are as above and $t_c \in \mathcal{T}_c$ is a clock dependent term.

The set of actions is denoted by Act.

A protocol is represented by a set of sequences of actions $\Pi = \sum_{i=1}^{n} \alpha_{1}^{i} \cdots \alpha_{n_{i}}^{i}$, where $\alpha_{j}^{i} = \ell_{j}^{i} \xrightarrow{g,\mathcal{R},\beta_{j}^{i}} \ell_{j+1}^{i}$ for some β_{j}^{i} with $j \in \{1,\ldots,n_{i}\}$. Here, the labels ℓ represent control points and \sum is the usual non-deterministic choice. This corresponds to the interleavings of a fixed set of sessions put in parallel.Let $\mathcal{R} \subseteq \mathcal{C}$ be a subset of clocks, $\delta \in \mathbb{R}_{\geq 0}$ a constant, $\nu : \mathcal{C} \longrightarrow \mathbb{R}_{\geq 0}$ a valuation for clocks, and let $t_{c} \in \mathcal{T}_{c}$ be a clock dependent term. We denote by $\nu[\mathcal{R}]$ the valuation obtained from ν by resetting all clocks in \mathcal{R} , i.e. $\nu[\mathcal{R}](c) = 0$ for any $c \in \mathcal{R}$ and $\nu[\mathcal{R}](c) = \nu(c)$ for any $c \notin \mathcal{R}$; $\nu + \delta$ denotes the valuation which advances all clocks by the same delay δ , i.e. $(\nu + \delta)(c) = \nu(c) + \delta$; and $t_{c}[\nu]$ is the term obtained from t_{c} by replacing all occurrences of c by the value of $\nu(c)$.

Definition 3 (Operational Semantics). A configuration of a protocol run is given by a tuple (σ, E, ν, ℓ) consisting of a substitution σ , a set of messages E, a valuation of clocks ν and a control point ℓ . The operational semantics is defined as a labelled transitional system over the set of configurations Conf. The transition relation

$$(\sigma, E, \nu, \ell) \xrightarrow{\alpha} (\sigma', E', \nu', \ell')$$

is defined as follows:

- **Output:** $\alpha = \ell_j^i \xrightarrow{g,\mathcal{R},!t} \ell_{j+1}^i$. Then, we have $(\sigma, E, \nu, \ell_j^i) \xrightarrow{\alpha} (\sigma, E', \nu', \ell_{j+1}^i)$, if $j \leq n_i$, $[\![g]\!]_{\nu,\sigma} = 1$, $\sigma' = \sigma$, $E' = E \cup \{t(\sigma \oplus \nu[\mathcal{R}])\}$ and $\nu' = \nu[\mathcal{R}]$. That is, sending the message t (provided that guard g is satisfied by the actual configuration) amounts to reset clocks in \mathcal{R} and adding t evaluated

with respect to the substitution σ and the valuation of clocks $\nu[\mathcal{R}]$, to the knowledge of the intruder.

- Input: $\alpha = \ell_j^i \xrightarrow{g,\mathcal{R},?t(\tilde{x})} \ell_{j+1}^i$. Then, we have $(\sigma, E, \nu, \ell_j^i) \xrightarrow{\alpha} (\sigma', E, \nu', \ell_{j+1}^i)$, if $j \leq n_i$ and there is $\rho \in \Gamma(\tilde{x})$ with $E\sigma \vdash t(\sigma \oplus \rho)$, $[\![g]\!]_{\nu,\sigma \oplus \rho} = 1$, $\sigma' = \sigma \oplus \rho$, and $\nu' = \nu[\mathcal{R}]$.

That is, \mathbf{t} corresponds to receiving any message, known to the intruder, that matches with $\mathbf{t}\sigma$ by a substitution ρ , such that g is satisfied by the pair ν , $\sigma \oplus \rho$; in addition, this action resets clocks in \mathcal{R} .

- **Time Passing:** $(\sigma, E, \nu, \ell_j^i) \xrightarrow{\delta} (\sigma, E, \nu + \delta, \ell_j^i)$, for any $\delta \in \mathbb{R}_{\geq 0}$. This action represents the passage of δ time units; passage of an arbitrary time is denoted by $\xrightarrow{\tau} = \bigcup_{\delta \in \mathbb{R}_{\geq 0}} \xrightarrow{\delta}$.

The initial configuration is given by a substitution σ_0 , a set of terms E_0 such that the variables in E_0 do not appear in the protocol description, a valuation ν_0 and a control point $\ell_0 \in \{\ell_0^1, \dots, \ell_{n_i}^n\}$.

Example 1. The Denning-Sacco shared key protocol [3], a protocol for distribution of a shared symmetric key by a trusted server and mutual authentication. Here, the timestamps are used to ensure the freshness of the shared key. Using the usual notation for cryptographic protocols, the protocol is described as follows:

$$\begin{array}{ll} A \rightarrow S : & A, B \\ S \rightarrow A : \{B, Kab, T, \{Kab, A, T\}_{Kbs}\}_{Kas} \\ A \rightarrow B : & \{Kab, A, T\}_{Kbs}\end{array}$$

We describe the protocol in our model. The constant parameters δ_1 , δ_2 represent network delays for *A* respectively *B*. We use a special clock *now* which is a global clock that is never reset and has an arbitrary initial value. For convenience of notation, we omit the control locations.

$$\begin{array}{l} A: !(A,B); \ now - T_1 < \delta_1 \to ?\{B,x,T_1,y\}_{smk(A,S)}; !y \\ B: now - T_2 < \delta_2 \to ?\{u,p,T_2\}_{smk(B,S)} \\ S: ?(z,v); !\{v,K,now,\{K,z,now\}_{smk(v,S)}\}_{smk(z,S)} \end{array}$$

Each participant of the protocol may be seen as a sequential process. As a sample we explain the actions of participant A. First, he sends his identity A and the identity of B to the server. Then, A waits to the message $\{B, x, T_1, y\}_{smk(A,S)}$. If T_1 , is a valid timestamp, i.e. the difference between the current time and the value of T_1 is less than the constant parameter δ_1 then A accepts x as session key and forwards the message y to B.

3 The TSPL Logic

In this section, we introduce the constraints/formulae we use to describe security properties. The logic we introduce allows to describe secrecy, authentication and any safety property.

Henceforth, let $K \subseteq \mathcal{K}$ be a fixed but arbitrary set of keys, such that $\emptyset \neq K \neq \mathcal{K}$ This set of keys can be understood as the set of keys whose inverses are secret.

3.1 Term Transducers and the Main Modality of the Logic

A pair $(\{t\}_k, r)$, where t is a term, $k \in K$ and r a critical position in $\{t\}_k$ is called a *term transducer (TT for short)*. Intuitively, the pair $(\{t\}_k, r)$ can be seen as function that takes as argument a term that matches with $\{t\}_k$ and returns as result the term $\{t\}_{k|r}$. As it will become clear later, a run of a cryptographic protocol provides the intruder with term transducers she (he) can apply to learn new terms.

The main modality of the logic we use can be defined as follows: Let m and s be two messages and let $w \in (\mathcal{M} \times \mathcal{P}os)^*$ be a sequence of term transducers. We define the predicate $m\langle w \rangle s$, which we read "s is w-protected in m", recursively on the structure of m and length of w:

- -m is atomic and $m \neq s$, or
- $-m = (m_1, m_2), m \neq s$ and both $m_1 \langle w \rangle s$ and $m_2 \langle w \rangle s$ are true, or
- $-m = \{m_1\}_k, m \neq s, k \notin K \text{ and } m_1 \langle w \rangle s \text{ is true, or }$
- $-m = \{m_1\}_k, m \neq s, k \in K \text{ and } w = \epsilon, \text{ or }$
- $(m-m = \{m_1\}_k, w = (b, r).w_1, m \neq s, k \in K, \text{ and } (m \neq b \text{ or } m|_r \langle w_1 \rangle s) \text{ is true.}$

This definition is easily generalized to sets of messages: Let M and S be sets of messages, w a sequence of term transducers and K a set of keys. We say that the secrets S are w - protected in M denoted by $M\langle w \rangle S$, if it holds $\bigwedge_{m \in M.s \in S} m\langle w \rangle s$.

Closure of Sets of Secrets. In this section, we define when a set of messages is closed. Closed sets of secrets enjoy the property that they are not derivable by composition. Intuitively, a set of messages is closed, if it contains, for any message m in the set, all messages along at least one path of the tree representing the message m.

Let M be a set of sets of messages and let m be a message. We use the notation: $m \odot M = \{M_i \cup \{m\} | M_i \in M\}$. The closure of a set S ensures that the intruder cannot derive a message in S by composition rules.

Definition 4 (Closure).

$$wc(m) = m \odot \left\{ egin{array}{ll} wc(m1) \cup wc(m2) & if \ m = (m1,m2) \ wc(m') \cup wc(k) & if \ m = \{m'\}_k \ \{K^{-1}\} & if \ m \ is \ atomic \end{array}
ight.$$

where $K^{-1} = \{k^{-1} \mid k \in K\}$. A set M of messages is called closed, if for any $m \in M$ there exists $M' \in wc(m)$ such that $M' \subseteq M$.

Example 2. Consider the message $m = (\{A, N\}_k, B)$. Then wc(m) consists of the following sets:

$$\begin{split} K^{-1} \cup \{(\{A,N\}_k,B),\{A,N\}_k,(A,N),A\} & K^{-1} \cup \{(\{A,N\}_k,B),\{A,N\}_k,(A,N),N\} & K^{-1} \cup \{(\{A,N\}_k,B),\{A,N\}_k,(A,N),N\} & K^{-1} \cup \{(\{A,N\}_k,B),B\}. \end{split}$$

We use the notation $E\langle w_i, S_i \rangle_I$ for $\bigwedge_{i \in I} E\langle w_i \rangle S_i$. Our purpose now is to define conditions on w_i and S_i such that for any set E of messages, if $E\langle w_i, S_i \rangle_I$ then $m\langle w_i, S_i \rangle_I$, for any message m derivable from E. In other words, such conditions ensure that $E\langle w_i, S_i \rangle_I$ is stable under the derivations rules defining the intruder.

Example 3. Let $E = \{s_1, s_2\}$ be a set of messages. Then we have $E\langle w \rangle(s_1, s_2)$. But we have both $E \vdash (s_1, s_2)$ and $\neg(s_1, s_2) \langle w \rangle(s_1, s_2)$.

This example shows that we need to consider only closed sets of secrets. But this is not sufficient, as showed by the following example.

Example 4. Let $E = \{\{s\}_{k_1}, k_2\}$ be a set of messages. We have $E\langle (\{\{s\}_{k_1}\}_{k_2}, 11)\rangle s$. But we have both $E \vdash \{\{s\}_{k_1}\}_{k_2}$ and $\neg \{\{s\}_{k_1}\}_{k_2} \langle (\{\{s\}_{k_1}\}_{k_2}, 11)\rangle s$.

TEAM LING

Hence, we need to deal also with the interior term transducers. To do so, let (b, p) be a term transducer. Then, we denote by lpt(b, p) the next term transducer in *b* from above that dominates $b_{|p}$, if it exists. For the sake of intuition, we skip the formal definition, and we give an example.

Example 5. Let b be the term $\{(\{N\}_{k'}, A)\}_k$ with $k, k' \in K$. Then, $lpt(b, 111) = (\{N\}_{k'}, 1)$. But lpt(b, 12) does not exist neither lpt(b, 11) does.

We have now everything we need to express the conditions that guarantee stability under the intruder's derivations:

Definition 5. $(w_i, S_i)_{i \in I}$ is called well-formed, if the following conditions are satisfied for every $i \in I$:

- 1. S_i is closed,
- 2. if $w_i = (b, r).w$, then the following conditions are satisfied:
 - if $w \neq \epsilon$, there exists $j \in I$ such that $w_j = w$ and $S_i \subseteq S_j$,
 - if there exists a term transducer $(b_1, r_1) = lpt(b, r)$, then there exists $k \in I$ such that either $b \in S_k$ or $w_k = (b_1, r_1) \cdot w$ and $S_i \subseteq S_k$.

The main property of $E\langle w_i, S_i \rangle_I$ is that it is stable under the intruder's deduction rules. Indeed, we have:

Proposition 1. Let E be a set of messages, and let $(w_i, S_i)_{i \in I}$ be well-formed such that $E\langle w_i, S_i \rangle_I$. Moreover, let m be a message with $E \vdash m$. Then, $m\langle w_i, S_i \rangle_I$.

Proposition 2. Let m be a message and E a set of messages such that $\mathcal{K} \setminus K^{-1} \subseteq E$. Then, $E \not\vdash m$ iff there exists a set of messages $A \in wc(m)$ s.t. $E\langle\epsilon\rangle A$.

3.2 TSPL: A Logic for Security Properties

In order to express general secrecy properties that involve variables, we introduce a new set of function symbols \mathcal{B} . Extended terms are build as before except that now we allow function symbols in \mathcal{B} to occur applied to variables, which we denote by x.f. Given a substitution σ that associates a message m to x, it will associate a set in wc(m) to x.f.

The syntax of TSPL is defined in the next table, where X is a fixed secondorder variable that ranges over sets of messages, x is a meta-variable that ranges over the set \mathcal{X} of first-order variables and f is a meta-variable that ranges over \mathcal{B} . Moreover, S is a finite set of extended terms and w is a finite sequence of term transducers that can contain free variables.

 $\begin{array}{ll} \Psi ::= \top \mid \sum_{i=1}^{n} a_i c_i + \sum_{j=1}^{m} b_j T_j \bowtie \delta \mid \Psi_1 \land \Psi_2 \mid \Psi_1 \lor \Psi_2 & \text{time constraints} \\ \Phi ::= X \langle w \rangle S \mid x \langle w \rangle S \mid t \langle \epsilon \rangle x.f \mid x = t \mid pc = \ell \mid \forall f \Phi \mid \top \mid \Phi \land \Phi \mid \neg \Phi & \text{term form.} \\ \Gamma ::= \Phi \mid \Psi \mid \Gamma \lor \Gamma \mid \Gamma \land \Gamma & \text{TSPL form.} \end{array}$

Notice that omitting the negation for time constraints is not essential as any negation of a time constraint can be put in a positive form.

Formulae are interpreted over a restricted set of configurations $Conf = \{(\sigma, E, \nu, l) | (\sigma, E, \nu, l) \in Conf, \mathcal{K} \setminus K^{-1} \subseteq E\}.$

Definition 6 (Semantics). The interpretation of a formula is given by the set of its models, i.e., the set $Conf_0$ of configurations that satisfy the formula. The definition is standard except for the following clauses:

$$\begin{split} \llbracket \Psi \rrbracket &= \left\{ (\sigma, E, \nu, \ell) \mid \llbracket \Psi \rrbracket_{\nu, \sigma} = 1 \right\} \\ \llbracket x = t \rrbracket &= \left\{ (\sigma, E, \nu, \ell) \mid \sigma(x) = \sigma(t) \right\} \\ \llbracket pc = \ell \rrbracket &= \left\{ (\sigma, E, \nu, \ell') \mid \sigma(x) = \sigma(t) \right\} \\ \llbracket t \langle \varphi \rangle S \rrbracket &= \left\{ (\sigma, E, \nu, \ell) \mid \{\sigma(x)\} \langle w\sigma \rangle S\sigma \right\} \\ \llbracket t \langle \varphi \rangle S \rrbracket &= \left\{ (\sigma, E, \nu, \ell) \mid \{\sigma(x)\} \langle w\sigma \rangle S\sigma \right\} \\ \llbracket t \langle \varphi \rangle S \rrbracket &= \left\{ (\sigma, E, \nu, \ell) \mid \{\tau\sigma\} \langle \varphi \rangle \langle x, f \sigma \rangle \right\}. \end{split}$$

For convenience of notations, we extend the set of formulae TSPL as follows:

$$TSPL_{+} \ni \varphi, \psi ::= \dots \mid (X, x) \langle w \rangle S \mid t \langle w \rangle S$$

The semantics of these formulae is:

 $\llbracket t \langle w \rangle S \rrbracket = \left\{ (\sigma, E, \nu, \ell) \mid t \sigma \langle w \sigma \rangle S \sigma \right\} \qquad \llbracket (X, x) \langle w \rangle S \rrbracket = \llbracket X \langle w \rangle S \rrbracket \cap \llbracket x \langle w \rangle S \rrbracket$

We can prove that any formulae of the form $t\langle w \rangle S$ is definable in TSPL. We use the notations $(\sigma, E, \nu, \ell) \models \varphi$ for $(\sigma, E, \nu, \ell) \in \llbracket \varphi \rrbracket$, $t\langle \psi \rangle S$ for $\neg t\langle w \rangle S$, $X\langle \psi \rangle S$ for $\neg X \langle w \rangle S$ and \bot for $\neg T$. Also, given a term *s*, we write $X \langle w \rangle s$ instead of $X \langle w \rangle \{s\}$ and $t\langle w \rangle s$ instead of $t\langle w \rangle \{s\}$. We identify formulae modulo the usual properties of boolean connectives such as associativity and commutativity of \land , \lor , distributivity etc... and use \Rightarrow as the classical logical implication (it can be easily defined in TSPL logic using set inclusion).

Well-Formed Formulae. We extend now the notion of closure of sets of messages to sets of extended terms. The definition is similar except that we have to consider two new cases: 1.) the case of a term t of the form $x.f: wc(t) = t \odot \{K^{-1}\}$ and 2.) the case of a variable $x: wc(x) = x.f \odot \{K^{-1}\}$, where f is a fresh function symbol. The Definition 5, that defines when $(w_i, S_i)_{i \in I}$ is well-formed for S_i sets of messages, is now easily extended to sets of extended terms. As now we are dealing with formulae, we have to define when a formula is well-formed in the same sense.

Definition 7. A formula Φ is well-formed, if for any sequence of term transducers w and closed set of terms S, whenever $\Phi \Rightarrow X\langle w \rangle S$, there exist $(w_i, S_i)_{i \in I}$ well-formed, such that $\Phi \Rightarrow \bigwedge_{i \in I} X\langle w_i \rangle S_i$ and $(w, S) \in (w_i, S_i)_{i \in I}$.

The main property satisfied by well-formed formulae is a parallel to Proposition 1 and is given by the following corollary, which is a direct consequence of Definitions 5 and 7.

Corollary 1. Let Φ be a well-formed formula such that $\Phi \Rightarrow X\langle w \rangle S$ and let $(\sigma, E, \nu, l) \in \llbracket \Phi \rrbracket$. If m is a message such that $E(\sigma \oplus \nu) \vdash m$, then $m \langle w(\sigma \oplus \nu) \rangle S(\sigma \oplus \nu)$.

Now, the property of Corollary 1 turns out to be crucial for developing a complete symbolic method and well-formedness has to be preserved. Therefore, we introduce the function \mathcal{H} . It takes as arguments a formula $X \langle b.w \rangle S$ and computes

the weakest (the largest w.r.t. set inclusion) well-formed formula $\mathcal{H}(X\langle b.w\rangle S)$, such that $\mathcal{H}(X\langle b.w\rangle S) \Rightarrow X\langle b.w\rangle S$:

$$\mathcal{H}(X\langle b.w\rangle S) = \begin{cases} X\langle b.w\rangle S \land \mathcal{H}(X\langle w\rangle S) & \text{if lpt}(b) \text{ is undefined} \\ X\langle b.w\rangle S \land \mathcal{H}(X\langle w\rangle S) \land \\ (\mathcal{H}(X\langle b_1.w\rangle S) \lor \bigvee_{S' \in wc(t)} X\langle \epsilon \rangle S') & \text{if } b = (t,p) \land b_1 = \mathrm{lpt}(b) \end{cases}$$

Proposition 3. Let Φ be a well-formed formula. Let b.w be a sequence of term transducers and S a closed set of extended terms such that $\Phi \Rightarrow X \langle b.w \rangle S$. Then $\Phi \Rightarrow \mathcal{H}(X \langle b.w \rangle S)$.

4 Computing Predecessors

We are interested in proving reachability properties of bounded timed cryptographic protocols. Given a property φ and an action α , $pre(\alpha, C)$ denotes the smallest set of configurations that by executing α may lead to a configuration that satisfies φ . That is,

Definition 8 (Predecessors). The predecessor of a set of configurations $C \subseteq Conf$ with respect to an action α , denoted $pre(\alpha, C)$ is the set of configurations s, such that there is at least one possible execution of α that leads from s to a configuration in C. More precisely

 $pre(\alpha, \mathcal{C}) ::= \big\{ (\sigma, E, \nu, l) \mid \exists (\sigma', E', \nu', l') \in \mathcal{C} \ s. \ t. \ (\sigma, E, \nu, l) \stackrel{\alpha}{\longrightarrow} (\sigma', E', \nu', l') \big\}.$

Given a formula Γ , we use $pre(\alpha, \Gamma)$ instead of $pre(\alpha, \llbracket \Gamma \rrbracket)$ to denote the predecessor of a formula $\Gamma \in \text{TSPL}$.

The purpose of this section is to show that $pre(\alpha, \Gamma)$ is effectively expressible in TSPL, when Γ is a positive boolean combination of time constraints and term formulae of the form:

$$\varPhi ::= x = t \mid pc = \ell \mid \top \mid \bot \mid x \neq t \mid pc \neq \ell \mid X \langle \mathbf{p} \rangle S \mid (X, x) \langle \mathbf{p} \rangle S \mid t \langle \mathbf{p} \rangle x.f \mid \forall f \varPhi.$$

First, it is easy to see that $pre(\alpha, \Phi) = \Phi$, if α is a time passing action and Φ is a term formula. Also, for any action $\alpha = l \xrightarrow{g,\mathcal{R},lt} l'$, respectively $\alpha = l \xrightarrow{g,\mathcal{R},?t} l'$, and any time constraint Ψ , we have $pre(\alpha, \Psi) = g \wedge pc = \ell \wedge \Psi[R]$.

Moreover, it can easily be shown that for the actions considered here *pre* distributes with respect to disjunction; in addition, if α is an output or an input, we can easily prove that *pre* distributes with conjunction too; we can also prove that *pre* commutes with \forall , i.e. $pre(\alpha, \forall f\Phi) = \forall f \ pre(\alpha, \Phi)$.

Time Passing and Time Constraints. In this section, we show that the predecessor of $\llbracket \Psi \rrbracket$, where Ψ is a time constraint, can be described by an TSPL formula. We consider the action $\xrightarrow{\tau}$, i.e. time passing. The case of input and output actions is described above.

We need first to define three kinds of normal forms for time constraints. Let Ψ be the atomic time constraint $\sum_{i=1}^{n} a_i c_i + \sum_{j=1}^{m} b_j T_j \bowtie d$. We denote by

 $\mathscr{C}(\Psi)$ the sum of the coefficients of clocks, i.e. $\sum_{i=1}^{n} a_i$. Then, an atomic time constraint $\Psi \equiv \sum_{i=1}^{n} a_i c_i + \sum_{j=1}^{m} b_j T_j \bowtie d$ is in *positive normal form* (PNF for short), if $\mathscr{C}(\Psi) > 0$; it is in *negative normal form* (NNF for short), if $\mathscr{C}(\Psi) < 0$; and finally, it is in *0-normal form*, if $\mathscr{C}(\Psi) = 0$.

Clearly any time constraint can be put in the form of a disjunction of conjunctions of the form $\Psi_1 \wedge \Psi_2 \wedge \Psi_3$, where Ψ_1 is a conjunction of formulae in PNF, Ψ_2 is a conjunction of formulae in NNF and Ψ_3 is a conjunction of formulae in 0-NF. For the rest of this section, we write $\psi \in \Psi_i$ to state that ψ is a conjunct of Ψ_i , i.e., we view conjunctions of formulae as sets of formulae.

Thus, let us consider a time constraint of the form $\Psi_1 \wedge \Psi_2 \wedge \Psi_3$ as above. Then, $pre(\xrightarrow{\tau}, \Psi_1 \wedge \Psi_2 \wedge \Psi_3)$ can be described by the formula $\exists \delta \geq 0 \cdot \Psi_1 + \delta \wedge \Psi_2 + \delta \wedge \Psi_3 + \delta$. We have then to show that we can eliminate the quantification on δ while obtaining a time constraint.

First, notice that $\Psi_3 + \delta$ is logically equivalent to Ψ_3 , since it is in 0-NF. Therefore, we can rewrite the formula to the equivalent formula $\exists \delta \ge 0 \cdot (\Psi_1 + \delta \land \Psi_2 + \delta) \land \Psi_3$ and focus on discussing how to transform $\exists \delta \ge 0 \cdot (\Psi_1 + \delta \land \Psi_2 + \delta)$ into an equivalent time constraint. Let us explain the main idea by considering the simple case where Ψ_1 and Ψ_2 are simple conjunctions.

The Simple Case. Consider a PNF time constraint $\Psi_1 \equiv \sum_{i=1}^n a_i c_i + \sum_{j=1}^m b_j T_j \bowtie_1 d$ and a NNF $\Psi_2 \equiv \sum_{i=1}^n a'_i c_i + \sum_{j=1}^m b'_j T_j \bowtie_2 d'$. Then, we have:

$$\begin{split} \Psi_1 + \delta &\equiv \sum_{i=1}^n a_i c_i + \sum_{j=1}^m b_j T_j + \delta \sum_{i=1}^n a_i \bowtie_1 d \\ \Psi_2 + \delta &\equiv \sum_{i=1}^n a_i' c_i + \sum_{j=1}^m b_j' T_j + \delta \sum_{i=1}^n a_i' \bowtie_2 d' \end{split}$$

By multiplying with $\mathscr{C}(\Psi_1)$ and $|\mathscr{C}(\Psi_2)|$ and adding the right-hands we get the following time constraint:

$$\sum_{i=1}^n a_i'' c_i + \sum_{j=1}^m b_j'' T_j \bowtie' |\mathscr{C}(\Psi_2)| d + \mathscr{C}(\Psi_1) d'$$

with $a_i'' = |\mathscr{C}(\Psi_2)|a_i + \mathscr{C}(\Psi_1)a_i', b_j'' = |\mathscr{C}(\Psi_2)|b_i + \mathscr{C}(\Psi_1)b_i'$ and if $\bowtie_1 \equiv \bowtie_2$ then $\bowtie' \equiv \bowtie_1$ else $\bowtie' \equiv \lt$.

Let us denote this formula by $\Delta(\Psi_1, \Psi_2)$. Notice that $\Delta(\Psi_1, \Psi_2)$ is independent of δ . One can prove that $\exists \delta \geq 0 \cdot (\Psi_1 + \delta \wedge \Psi_2 + \delta)$ is equivalent to the time constraint $\Psi_1 \wedge \Delta(\Psi_1, \Psi_2)$. The conjunct Ψ_1 has to be kept as we are interesting in the predecessors, thus the upper bound on the clocks must be satisfied as time only increases.

The General Case. Let us now return to the general case, where Ψ_1 and Ψ_2 are arbitrary conjunctions of formulae in PNF, respectively, NNF. To handle this case we generalize Δ to sets (conjunctions of formulae as follows):

1. $\Delta(\emptyset, \psi) = \top$ 2. $\Delta(\psi, \emptyset) = \psi$ 3. $\Delta(\Psi_1, \Psi_2) = \bigwedge_{\psi_1 \in \Psi_1, \psi_2 \in \Psi_2} \Delta(\psi_1, \psi_2)$

Then we can prove that $\exists \delta \geq 0 \cdot (\Psi_1 + \delta \wedge \Psi_2 + \delta)$ is equivalent to $\Delta(\Psi_1, \Psi_2)$.

Summarizing together, we can transform $\exists \delta \geq 0 \cdot \Psi_1 + \delta \wedge \Psi_2 + \delta \wedge \Psi_3 + \delta$ into the equivalent time constraint $\Delta(\Psi_1, \Psi_2) \wedge \Psi_1 \wedge \Psi_3$. Hence, if we define $\operatorname{Pre}(\xrightarrow{\tau}, \Psi_1 \wedge \Psi_2 \wedge \Psi_3) \stackrel{def}{=} \Delta(\Psi_1, \Psi_2) \wedge \Psi_1 \wedge \Psi_3$, we obtain the following result:

Proposition 4. For any time constraint Ψ , $pre(\xrightarrow{\tau}, \llbracket \Psi \rrbracket) = \llbracket Pre(\xrightarrow{\tau}, \Psi) \rrbracket$.

Output Action and Atomic Term Formulae. Throughout this section let $\alpha = l \xrightarrow{g,\mathcal{R},lt} l'$, and let \tilde{c} be all the clocks that occur in t and do not occur in \mathcal{R} . We show that we can express $pre(\alpha, \varphi)$, for any atomic term formula φ . The core point here is how we deal with the clocks occurrences in the sent message. Since the values of clocks change with time, we have to freeze these values in the message added to the intruder knowledge; we do this by replacing in t, all occurrences of clocks that are not reseted \tilde{c} with fresh time variables \tilde{T}_c and by introducing the constraints $\tilde{T}_c = \tilde{c}$. For more details, see the example presented in Appendix A.

Let us define $\mathbf{Pre}(\alpha, \varphi)$:

- 1. **Pre** $(\alpha, \varphi) \stackrel{\text{def}}{=} g \wedge pc = \ell \wedge (\varphi \vee ((X, t[0/\mathcal{R}, \tilde{T}_c/\tilde{c}])\langle \psi \rangle S \wedge \tilde{T}_c = \tilde{c}))$, where \tilde{T}_c are fresh time variables, if φ is a formula of the form $X\langle \psi \rangle S$ or $(X, x)\langle \psi \rangle S$.
- 2. **Pre** $(\alpha, \varphi) \stackrel{def}{=} g \wedge pc = \ell \wedge \varphi$, if φ is of the form $x \neq t', x = t', \top, \bot, pc = \ell'$ or $t(\not e) x.f$.

Proposition 5. For any output action α and atomic term formula φ , $pre(\alpha, \llbracket \varphi \rrbracket) = \llbracket \operatorname{Pre}(\alpha, \varphi) \rrbracket$.

Input Action and Atomic Term Formulae. Throughout this section let $\alpha = l \xrightarrow{g,\mathcal{R},?t(\tilde{x})} l'$. We show that we can express $pre(\alpha,\varphi)$, for any atomic term formula φ . To do so, we need to introduce a few definitions and prove a few intermediate results.

Let t be a term and p a critical position in t. Then, we denote by lpp(t, p) the position of the first term transducer in t from above that dominates p if it exists.

Example 6. Consider the term $t = (\{A, \{N\}_{k_1}\}_{k_2}, N)$, where $k_1, k_2 \in K$. Let p = 1121 and p' = 2. Thus, $t_{|p|} = t_{|p'|} = N$. Then, we have lpp(t, p) = 1, which corresponds to the key k_2 ; lpp(t, p') is, however, undefined.

Given a term t, let $\mathbf{F}(t)$ denote the formula $\forall \vec{f} \bigwedge_{\substack{S' \in wc(t)}} X \langle \not e \rangle S'$ where \vec{f} is the

set of all fresh variables $f \in \mathcal{BX}$ that occur in wc(t). The intuitive explanation of next lemma is the following: being in a state (σ, E, ν, l) , in order to be able to make an input $t(\tilde{x})$, such that \tilde{x} are instantiated by ρ , it must be that $(\sigma, E, \nu, l) \in [F(t\rho)]$.

Lemma 1. Let *E* be a set of terms, *l* be a label, ν be a clocks valuation and let ρ and σ be ground substitutions such that $dom(\rho) = \tilde{x}$ and $(dom(\sigma) \cup var(E)) \cap \tilde{x} = \emptyset$. Then it holds $(\sigma, E, \nu, l) \in [F(t\rho)]$ iff $E\sigma \vdash t(\sigma \oplus \rho)$.

First, notice that the effect of an input action ?t depends on the messages that match with t and that are known by the intruder. Therefore, we need to characterize the set of configurations s, such that if in the next step x is instantiated by an input $?t(\tilde{x})$, the reached configuration s' satisfies $x\langle \psi \rangle S$. To understand how this characterization is obtained, the best is to consider the negation of $x\langle \psi \rangle S$, i.e., $x\langle w \rangle S$. The key idea can be explained by considering the sequence of actions $?t(\tilde{x})$; !x. That is, if a secret s that appears in x has to be protected then it has to appear in x under an encryption. Thus, before executing $?t(\tilde{x})$; !x, it should be the case that if we provide the intruder with the term transducer that takes as input $t(\tilde{x})$ and yields x, it is not possible to derive s.

Lemma 2. Let t be a term, S a set of terms, w a sequence of term transducers, x a variable and $P_{x,t}$ the set of critical positions of x in t. Let

$$\mathcal{K}(t,x,w,S) = X\langle w \rangle S \wedge \bigwedge_{p=lpp(t,p_x), p_x \in P_{x,t}} \mathcal{H}(X\langle (t|_p, p^{-1}p_x).w \rangle S).$$

Let E be a set of terms, l and l' labels, and ρ , σ ground substitutions such that $dom(\rho) = \tilde{x}, x \in \tilde{x}, (dom(\sigma) \cup var(E)) \cap \tilde{x} = \emptyset$. Let Φ a well-formed formula such that whenever $E\sigma \vdash t(\sigma \oplus \rho)$, we have $(\sigma \oplus \rho, E, l') \in [(X, x) \langle w \rangle S]$ iff $(\sigma, E, l) \in [\![\Phi]\!]$. Then $[\![\Phi]\!] = [\![\rho(\mathcal{K}(t, x, w, S))]\!]$.

Let now α be the action $\alpha = l \xrightarrow{g,\mathcal{R},?t(\tilde{x})} l'$, where \tilde{x} the variables that are instantiated by this action. We then define **Pre** (α, φ) as follows:

- 1. **Pre** $(\alpha, \varphi) \stackrel{def}{=} g \wedge pc = \ell \wedge F(t) \wedge \varphi$, if φ is of the form $X\langle \psi \rangle S$, $(X, y)\langle \psi \rangle S$, $t\langle \varphi \rangle x.f, x \neq t', x = t', pc = \ell', \top$ or \bot and $y \notin \tilde{x}$.
- 2. $\mathbf{Pre}(\alpha, (X, x) \langle \mathscr{W} \rangle S) \stackrel{def}{=} g \wedge pc = \ell \wedge F(t) \wedge \neg \mathcal{K}(t, x, w, S), \text{ if } x \in \tilde{x}.$

Proposition 6. For any input action α and atomic term formula φ , $pre(\alpha, \llbracket \varphi \rrbracket) = \llbracket Pre(\alpha, \varphi) \rrbracket$.

Collecting the results together, it is easy to see that for any formula $\varphi \in \text{TSPL}$ and any action α , $\text{Pre}(\alpha, \varphi) \in \text{TSPL}$. Then, we have the following theorem:

Theorem 1. Let α be any action and φ any formula in TSPL. Then, $pre(\alpha, \llbracket \varphi \rrbracket) = \llbracket \operatorname{Pre}(\alpha, \varphi) \rrbracket$.

5 Decidability of TSPL

In this section, we give a procedure for the decidability of the existence of a model of a TSPL formula. Notice that since we showed in Section 4 that given a formula φ in TSPL and a bounded CP π , one can compute **Pre**(π , φ), decidability of the satisfiability of formulae yields a decision procedure for reachability of configurations described by TSPL formulae.

To prove decidability for the satisfiability of TSPL formulae we follow a rule based approach (e.g., [12, 5] for two nice surveys) i.e.:

- 1. We introduce a set of formulae in *solved form*. For these formulae it is easy to decide whether a model exists.
- 2. We introduce a set of formulae in *intermediate form*. For each formula in intermediate form, we show how to reduce its satisfiability to the satisfiability of a set of *saturated formulae* in intermediate form; moreover, for each saturated formula in intermediate form, we can extract a formula in solved form such that a model exists for the formula in intermediate form if and only if the extracted formula in solved form is satisfiable.
- 3. We introduce a set of rewriting rules to transform any formula in the existential fragment into an intermediate form.
- 4. We prove soundness and completeness of these rules.
- 5. We also prove their termination for a given control, i.e. that normal forms are reached and that normal forms are indeed in intermediate form.

Then, we obtain the following result:

Theorem 2. Satisfiability for TSPL is decidable.

6 Conclusions

In this paper, we have proved the decidability of a large class of reachability properties, including secrecy and authentication, for **timed** bounded protocols. Our model for specifying timed protocols uses clocks, time variables and time-stamps. This work can be extended in several ways: 1.) our model can be naturally extended to associate time values to short term keys such that if the intruder obtains a message encrypted by a short term key then after the specified amount of time elapses the key becomes known by the intruder. Our model can and verification method can be extended to handle this model; 2.) our model can also be extended handle drifting clocks. It is well-known that models with clocks with drifts in bounded intervals can be transformed into models with perfect clocks modulo an abstraction, that is, taking into account more behavior. As discussed by Gong [10] drifting clocks can add subtle attacks; 3.) in the full version of this paper we show how we can use our logic to device an abstract interpretation based method for unbounded protocols.

References

- G. Bella and L. C. Paulson. Mechanizing BAN Kerberos by the inductive method. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV'98)*, pages 416–427, Vancouver, B.C., Canada, June 1998. Springer-Verlag LNCS 1427.
- 2. L. Bozga, C. Ene, and Y. Lakhnech. On the existence of an effective and complete proof system for bounded security protocols. In *FOSSACS*, vol. 2987. LNCS, 2004.
- 3. J. Clark and J. Joacob. A survey on authentication protocol. Available at the url http://www.cs.york.ac.uk/~jac/papers/drareviewps.ps, 1997.

- 4. Ernie Cohen. Taps: A first-order verifier for cryptographic protocols. In *Proceedings* of the 13th IEEE Computer Security Foundations Workshop (CSFW'00), page 144. IEEE Computer Society, 2000.
- 5. H. Comon. Disunification: A survey. In *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, Cambridge, MA, 1991.
- 6. G. Delzanno and P. Ganty. Automatic verification of time sensitive cryptographic protocols. In *TACAS: International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, LNCS,* volume 2988, 2004.
- D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions* on Information Theory, 29(2):198–208, 1983.
- Neil Evans and Steve Schneider. Analyzing time dependent security properties in CSP using PVS. In *ESORICS*, pages 222–237, 2000.
- M. Fiore and M. Abadi. Computing symbolic models for verifying cryptographic protocols. In 14th IEEE Computer Security Foundations Workshop (CSFW '01), pages 160–173, Washington - Brussels - Tokyo, June 2001. IEEE.
- 10. Li Gong. A security risk of depending on synchronized clocks. *Operating Systems Review*, 26(1):49–53, 1992.
- Roberto Gorrieri and Fabio Martinelli. A simple framework for real-time cryptographic protocol analysis with compositional proof rules. *Science of Computer Programming*, 50(1-3):23–49, 2004.
- J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: A rulebased survey of unification. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*. MIT-Press, 1991.
- G. Lowe. Breaking and fixing the Needham-Schroeder Public-Key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of LNCS, pages 147–166, 1996.
- 14. G. Lowe. A hierarchy of authentication specifications. In *Proc. of The 10th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.
- J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In ACM Conference on Computer and Communications Security, pages 166–175, 2001.
- M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. In *IEEE Computer Security Foundations Workshop*, 2001.
- 17. J. Thayer, J. Herzog, and J. Guttman. Honest Ideals on Strand Spaces. In *IEEE Computer Security Foundations Workshop*, pages 66–78, 1998.

A Computation of Predecessors for a Sequence of Actions

Let us give an example that shows how we compute the set of predecessors with respect to a simple protocol. Let $\alpha_0 = l_0 \xrightarrow{\top, \emptyset, ! \{c\}_k} l_1$, $\alpha_1 = l_1 \xrightarrow{\top, \{d\}, ! \{c\}_k} l_2$, $\alpha_2 = l_2 \xrightarrow{g_2, \emptyset, ? \{T\}_k} l_3$, $\alpha_3 = l_3 \xrightarrow{\top, \emptyset, ! s} l_4$ where *c* and *d* are clocks, *k* is a symmetric key (intended to remain secret for the intruder), *s* is a message (the secret) and $g_2 \equiv d = 1 \land -c + T < -1$. Let $\Phi \stackrel{def}{=} X \langle f \rangle s$ be the formula that represents the "bad configurations" (where secret *s* is known to the intruder). Now let $\Pi_1 = \alpha_1 \alpha_2 \alpha_3$ and $\Pi_0 = \alpha_0 \alpha_1 \alpha_2 \alpha_3$ are two protocols. We show that Π_1 is secure

w.r.t. to formula Φ , while the same assertion does not hold for Π_2 . For sake of simplicity, we work modulo \equiv .

 $\mathbf{Pre}(\xrightarrow{\tau}, X\langle \not e \rangle s) = X\langle \not e \rangle s.$ $\mathbf{Pre}(\alpha_3, X\langle \mathbf{f} \rangle s) = \top \land pc = l_3 \land (X, \{s\} \langle \mathbf{f} \rangle s \lor (s \langle \mathbf{f} \rangle s) \equiv pc = l_3$ $\mathbf{Pre}(\xrightarrow{\tau}, pc = l_3) = pc = l_3$ $\operatorname{Pre}(\alpha_2, pc = l_3) = \Phi_1$ where $\varPhi_1 \equiv d = 1 \land -c + T < -1 \land pc = l_2 \land X\langle \not e \rangle \{\{T\}_k, T\} \land X\langle \not e \rangle \{\{T\}_k, k\}$ $\mathbf{Pre}(\xrightarrow{\tau}, \Phi_1) = \Phi_2$ where $\Phi_2 \equiv (d \le 1) \land d - c + T < 0 \land pc = l_2 \land X \langle \not e \rangle \{\{T\}_k, T\} \land X \langle \not e \rangle \{\{T\}_k, k\}$ $\mathbf{Pre}(\alpha_1, \Phi_2) = \Phi_3$ where $\Phi_3 \equiv (0 \le 1) \land 0 - c + T < 0 \land pc = l_1 \land$ $(X, \{T_c\}_k) \langle \not e \rangle \{\{T\}_k, T\} \land (X, \{T_c\}_k) \langle \not e \rangle \{\{T\}_k, k\} \land T_c = c$ $\equiv -c+T < 0 \land pc = l_1 \land T_c = c \land T < T_c \land ((X \not e) \{\{T\}_k, T\} \land X \not e) \{\{T\}_k, k\}) \lor T_c = T)$ $\equiv -c + T < 0 \land pc = l_1 \land T_c = c \land T < T_c \land (X \not\in \{T\}_k, T\} \land X \not\in \{T\}_k, k\})$ $\mathbf{Pre}(\xrightarrow{\tau}, \Phi_3) = \Phi_4$ where $\Phi_4 \equiv pc = l_1 \land c \leq T_c \land T < T_c \land X \langle \not e \rangle \{ \{T\}_k, T\} \land X \langle \not e \rangle \{ \{T\}_k, k \}$ $\mathbf{Pre}(\alpha_0, \varPhi_4) = \varPhi_5 \text{ where } \varPhi_5 \equiv pc = l_0 \land c \leq T_c \land T < T_c \land (X, \{T'_c\}_k) \langle \forall \{\{T\}_k, T\} \land$ $(X,\{T_c'\}_k)\langle \not\!\!\! e\rangle \{\{T\}_k,k\}\wedge T_c'=c\equiv$ $pc = l_0 \wedge c \leq T_c \wedge T < T_c \wedge T_c' = c \wedge T_c' < T_c \wedge ((X \langle \not c \rangle \{\{T\}_k, T\} \wedge X \langle \not c \rangle \{\{T\}_k, k\}) \vee (X \wedge C \wedge T_c' = C \wedge T_c' < T_c \wedge ((X \wedge f \wedge T_c') \wedge T_c') \wedge (X \wedge f \wedge T_$ $T_c' = T$ $\mathbf{Pre}(\xrightarrow{\tau}, \phi_5) = \phi_6 \text{ where } \phi_6 \equiv pc = l_0 \land c \leq T_c \land c \leq T'_c \land T < T_c \land T'_c < T_c$ $\wedge ((X\langle \not e \rangle \{\{T\}_k, T\} \land X\langle \not e \rangle \{\{T\}_k, k\}) \lor T'_c = T)$

Hence, we obtain the following predecessors: $pre(\Pi_1, X \langle \mathbf{e} \rangle s) \equiv pc = l_1 \land c \leq T_c \land T < T_c \land X \langle \mathbf{e} \rangle \{\{T\}_k, T\} \land X \langle \mathbf{e} \rangle \{\{T\}_k, k\}$ $pre(\Pi_0, X \langle \mathbf{e} \rangle s) \equiv pc = l_0 \land c \leq T_c \land c \leq T'_c \land T < T_c \land T'_c < T_c \land$ $((X \langle \mathbf{e} \rangle \{\{T\}_k, T\} \land X \langle \mathbf{e} \rangle \{\{T\}_k, k\}) \lor T'_c = T)$

Since we supposed that k is a secret symmetric key (i.e. $X \langle f \rangle k$), if there is no any message of the form $\{T''\}_k$ known initially to the intruder, the protocol Π_1 is secure with respect to the secrecy of s. On the contrary, protocol Π_0 is unsecure. If we pick T_c , and T'_c such that $T < T_c \wedge T'_c < T_c \wedge T'_c = T$, then we obtain an attack, that corresponds to the fact that the first message sent by our participant can be replayed successfully by the intruder (it satisfies the time constraints), while the same is not true for the second sent message.

Deciding Probabilistic Bisimilarity Over Infinite-State Probabilistic Systems*

Tomáš Brázdil, Antonín Kučera, and Oldřich Stražovský

Faculty of Informatics, Masaryk University, Botanická 68a, 60200 Brno, Czech Republic {brazdil,tony,strazovsky}@fi.muni.cz

Abstract. We prove that probabilistic bisimilarity is decidable over probabilistic extensions of BPA and BPP processes. For normed subclasses of probabilistic BPA and BPP processes we obtain polynomialtime algorithms. Further, we show that probabilistic bisimilarity between probabilistic pushdown automata and finite-state systems is decidable in exponential time. If the number of control states in PDA is bounded by a fixed constant, then the algorithm needs only polynomial time.

1 Introduction

Theory of probabilistic systems is a formal basis for modeling and verification of systems that exhibit some kind of uncertainty [20, 18]. For example, this uncertainty can be caused by unpredictable errors (such as message loss in unreliable channels), randomization (as in randomized algorithms), or simply underspecification in some of the system components. The semantics of probabilistic systems is usually defined in terms of homogeneous Markov chains or Markov decision processes. The former model allows to specify just probabilistic behavioural aspects, while the latter one combines the paradigms of nondeterministic and probabilistic choice. In this paper we consider a generalized model of [23] which subsumes both of the aforementioned formalisms and also "ordinary" non-probabilistic systems. As we shall see, this means that the majority of our results generalize the ones which were previously established for non-probabilistic infinite-state systems.

Methods for formal verification of probabilistic systems follow the two standard approaches of *model-checking* and *equivalence-checking*. In the modelchecking approach, desired properties of the system are specified as a formula of a suitable probabilistic temporal logic (such as PCTL or PCTL* [7]), and then it is shown that the system satisfies the formula. In the equivalence-checking approach, one proves that the verified system is semantically equivalent to its *specification*, which is another probabilistic system. Here the notion of semantic equivalence can be formally captured in many ways. Most of the existing equivalences are probabilistic extensions of their non-probabilistic counterparts. One

^{*} The work has been supported by the Grant Agency of the Czech Republic, grant No. 201/03/1161.

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 193-208, 2004.

consequence of this is that various variants of *probabilistic bisimilarity* [21] play a very important role in this setting.

The state of the art: Algorithmic support for formal verification of probabilistic systems has so far been limited to finite-state systems [11, 15, 3, 12, 16, 6, 20, 13, 10]. Only recently, model-checking algorithms for infinite-state models of fully probabilistic lossy channel systems [17, 5, 1, 2, 22] and fully probabilistic pushdown automata [14] appeared. However, the authors are not aware of any results about equivalence-checking with probabilistic infinite-state systems.

Our Contribution: In the first part of our work we consider probabilistic extensions of the well-known families of BPA and BPP processes, which are denoted pBPA and pBPP, respectively. We have chosen a general extension based on the idea that process constants have finitely many basic transitions of the form $X \to \mu$ where μ is a probability distribution over pairs of the form (a, α) , where a is an action and α a sequence of BPA/BPP constants (in the case of BPP, sequences of constants are considered modulo commutativity and thus the concatenation operator models a simple form of parallel composition without synchronization). Basic transitions then define transitions performable from sequences of constants by adjusting the target distributions accordingly. Hence, our model subsumes the original (non-probabilistic) BPA and BPP, which can be understood as those subclasses of pBPA and pBPP where all distributions used in basic transitions are Dirac. Moreover, pBPA also subsumes a fully probabilistic extension of BPA. We prove that probabilistic bisimilarity (both in its combined and non-combined variant) is decidable for pBPA and pBPP processes. Moreover, for normed subclasses of pBPA and pBPP we have polynomial-time algorithms. Our results generalize the ones for non-probabilistic BPA and BPP by extending and adapting the original notions and proofs. Intuitively, such an extension is possible because probabilistic bisimilarity has similar algebraic and transfer properties as "ordinary" non-probabilistic bisimilarity. These properties can be reformulated and reproved in the probabilistic setting by incorporating some ideas for finite-state systems (e.g., the use of geometrical algorithms for finitely-generated convex spaces in the style of [10]), and there are also new techniques for handling problems which are specific to infinite-state probabilistic systems. After reestablishing these crucial properties, we can basically follow the original proofs because they mostly rely just on algebraic arguments. This can be seen as a nice evidence of the robustness of the original ideas.

In Section 4 we concentrate on checking probabilistic bisimilarity between processes of probabilistic pushdown automata (pPDA) and probabilistic finitestate automata. Our results are based on a generic method for checking semantic equivalences between PDA and finite-state processes proposed in [19]. This method clearly separates generic arguments (applicable to every behavioral equivalence which is a right PDA congruence in the sense of Definition 31) from the equivalence-specific parts that must be supplied for each behavioral equivalence individually. This method works also in the probabilistic setting, but the application part would be unnecessarily long and complicated if we used the original scheme of [19]. Therefore, the generic part of the method is first adjusted into a more "algebraic" form which simplifies some of the crucial steps. The method is then used to prove that probabilistic bisimilarity is decidable between pPDA and finite-state processes in exponential time. Actually, this algorithm is *polynomial* if the number of pPDA control states is bounded by a fixed constant (in particular, this holds for pBPA).

In all sections we tried to avoid repeating of the known things as much as possible; unfortunately, this inevitably means that the material is not completely self-contained. We did our best to provide enough information and intuition so that our presentation is understandable even for a reader who is not familiar with "classical" results on BPA and BPP presented in [9], and who does not know anything about the recent results of [19]. We always clearly mark the results which are not to be considered as a part of this work.

The results presented in this paper generate many questions. Some of them are summarized in Section 5. Due to space constraints, most proofs had to be omitted. These can be found in a full version of this paper [8].

2 **Basic Definitions**

We start by recalling basic notions of probability theory. A *discrete probability* measure (or *distribution*) over a set X is a function $\mu : 2^X \to \mathbb{R}^{\geq 0}$ such that, for each countable collection $\{X_i\}_{i \in I}$ of pairwise disjoint subsets of $X, \mu(\bigcup_{i \in I} X_i) =$ $\sum_{i \in I} \mu(X_i)$, and moreover $\mu(X) = 1$. The set of all distributions over a set X is denoted *Disc(X)*. A *Dirac* distribution is a distribution which assigns 1 to exactly one object. A *rational* distribution is a distribution which assigns a rational number to each object. For every $\mu \in Disc(X)$ we define its *support*, denoted $supp(\mu)$, as the set $\{x \in X \mid \mu(x) > 0\}$. A *discrete probability space* is a pair (X, μ) where X is a set called *sample space* and μ a distribution over X.

The underlying semantics of probabilistic systems is usually defined in terms of labelled Markov chains or labelled Markov decision processes, depending mainly on whether the considered system is sequential or parallel. Since some of our results are applicable to both sequential and parallel probabilistic systems, we use a more general formalism of [23] which subsumes the aforementioned models.

Definition 1. An action-labelled probabilistic transition system (*or just* transition system) is a triple S = (S, Act, D) where S is a finite or countably infinite set of states, $Act \neq \emptyset$ is a set of actions, and $D \subseteq S \times Disc(Act \times S)$ is a finite or countably infinite transition relation. An element $(s, \mu) \in D$ is called a transition and alternatively denoted by $s \rightarrow \mu$. A (probabilistic) process is a state of some transition system.

For the rest of this section, let us fix a probabilistic transition system S = (S, Act, D).

We say that $t \in S$ is reachable from $s \in S$ under a word $w = a_1 \cdots a_k \in Act^*$, written $s \xrightarrow{w} t$ (or simply $s \xrightarrow{*} t$ if w is irrelevant), if there is a finite sequence $s = s_0, s_1, \ldots, s_k = t$ of states such that $(s_i, \mu_i) \in D$ and $\mu_i(a_{i+1}, s_{i+1}) > 0$ for each $0 \le i < k$. For each transition $s \to \mu$ we define the set of μ -successors of s by $succ(s, \mu) = \{t \in S \mid \mu(a, t) > 0 \text{ for some } a \in Act\}$. For each state s we define the set of successors by $succ(s) = \bigcup_{s \to \mu} succ(s, \mu)$.

define the set of successors by $succ(s) = \bigcup_{s \to \mu} succ(s, \mu)$. For every $s \in S$, let $D(s) = \{(s, \mu) \in D\}$ be the set of transitions that leave from s. Every distribution $\sigma \in Disc(D(s))$ determines a unique distribution $\mu_{\sigma} \in Disc(Act \times S)$ defined for each $(a,t) \in Act \times S$ as $\mu_{\sigma}(a,t) = \sum_{(s,\mu)\in D(s)} \sigma(s,\mu)\mu(a,t)$. Note that the sum $\sum_{(s,\mu)\in D(s)} \sigma(s,\mu)\mu(a,t)$ exists because the set D(s) is finite or countably infinite. A combined transition relation $D_C \subseteq S \times Disc(Act \times S)$ is defined by $D_C = \{(s,\mu_{\sigma}) \mid s \in S, \sigma \in Disc(D(s))\}$. We write $s \to_C \mu$ instead of $(s,\mu) \in D_C$. Obviously, introducing combined transitions does not influence the reachability relation. However, a single state can have uncountably many outgoing combined transition system in the sense of Definition 1.

Semantic equivalence of probabilistic processes can be formally captured in many ways. Existing approaches extend the ideas originally developed for non-probabilistic processes, and the resulting notions have similar properties as their non-probabilistic counterparts. One consequence of this is that probabilistic extensions of *bisimulation-like equivalences* play a very important role in this setting. First we introduce some useful notions and notation. For the rest of this section, let us fix a transition system S = (S, Act, D). Let $E \subseteq S \times S$ be an equivalence relation. We say that two distributions $\mu, \nu \in Disc(Act \times S)$ are *equivalent according to E*, denoted $\mu E\nu$, iff for each $a \in Act$ and each equivalence class $C \in S/E$ we have that $\mu(a, C) = \nu(a, C)$, where $\mu(a, C) = \sum_{s \in C} \mu(a, s)$. In other words, the equivalence *E* (defined on states) determines a unique equivalence on distributions that is also denoted by *E*.

Definition 2. Let E be an equivalence on S, and let $(s,t) \in S \times S$. We say that (s,t) expands in E iff

- for each $s \to \mu$ there is $t \to \nu$ such that $\mu E\nu$; - for each $t \to \mu$ there is $s \to \nu$ such that $\mu E\nu$.

A relation $R \subseteq S \times S$ expands in E iff each $(s,t) \in R$ expands in E. An equivalence E on S is a probabilistic bisimulation iff E expands in E. We say that $s,t \in S$ are bisimilar, written $s \sim t$, iff they are related by some probabilistic bisimulation.

The notions of combined expansion, combined bisimulation, and combined bisimilarity (denoted \sim_C), are defined in the same way as above, using \rightarrow_C instead of \rightarrow .

It can be shown that probabilistic bisimilarity is a proper refinement of combined probabilistic bisimilarity (we refer to [23] for a more detailed comparison of the two equivalences). Since most of our results are valid for both of these equivalences, we usually refer just to "bisimilarity" and use the \rightarrow and \simeq symbols to indicate that a given construction works both for \rightarrow and \sim , and for \rightarrow_C and \sim_C , respectively. The word "expansion" is also overloaded in the rest of this paper.

Bisimilarity can also be used to relate processes of different transition systems by considering bisimulations on the disjoint union of the two systems.

Given a binary relation R over a set X, the symbol \equiv_R denotes the least equivalence on X subsuming R. We start with a sequence of basic observations.

Lemma 3. Let R_1, R_2 be binary relations on S such that $R_1 \subseteq R_2$. Then for all $\mu, \nu \in Disc(Act \times S)$ we have that if $\mu \equiv_{R_1} \nu$, then also $\mu \equiv_{R_2} \nu$.

Lemma 4. Let R be a relation on S and E be an equivalence on S. If R expands in E, then \equiv_R expands in E.

An immediate corollary to the previous lemmas is the following:

Corollary 5. \simeq is a bisimulation.

Proof. \simeq expands in \equiv_{\simeq} by Lemma 3, hence \equiv_{\simeq} expands in \equiv_{\simeq} by Lemma 4. Therefore, \equiv_{\simeq} is a bisimulation and $\equiv_{\simeq} \subseteq \simeq$.

Lemma 6. Suppose that $(s,t) \in E$ where E is a bisimulation on S. If $s \xrightarrow{w} s'$ for some $w \in Act^*$, then there is $t \xrightarrow{w} t'$ such that $(s',t') \in E$.

2.1 Approximating Bisimilarity

Bisimilarity can be approximated by a family of equivalences \simeq_i , $i \in \mathbb{N}_0$, defined inductively as follows:

 $-\simeq_0 = S \times S;$

 $- \simeq_{i+1}$ consists of those $(s,t) \in \simeq_i$ which expand in \simeq_i .

Clearly $\simeq \subseteq \bigcap_{i=0}^{\infty} \simeq_i$, and the other inclusion holds if each process $s \in S$ is *finitely branching*, i.e., the set $\{\mu \mid s \to \mu\}$ is finite. It is worth mentioning that this observation can be further generalized.

Lemma 7. Let $s, t \in S$, and let us assume that each state t' reachable from t is finitely branching (i.e., s can still be infinitely-branching). Then $s \simeq t$ iff $s \simeq_i t$ for each $i \in \mathbb{N}_0$.

Lemma 7 can be seen as a generalization of a similar result for nonprobabilistic processes and strong bisimilarity presented in [4]. Also note that Lemma 7 does not impose any restrictions on distributions which can have an infinite support.

Definition 8. We say that a process $s \in S$ is well-defined if s is finitely branching and for each transition $s \to \mu$ we have that μ is a rational distribution with a finite support.

For example, pBPA, pBPP, and pPDA processes which are introduced in next sections are well-defined.

Lemma 9. Let us assume that Act is finite, and let $s,t \in S$ be well-defined states. Let E be an equivalence over $succ(s) \cup succ(t)$ (represented as a finite set of its elements). The problem if (s,t) expands in E^1 is decidable in time polynomial in |D(s)| + |D(t)|. Here

$$|D(s)| = \sum_{s \to \mu} \sum_{\substack{(a,t) \in Act \times S \\ \mu(a,t) > 0}} |(\mu(a,t), a, t)|$$

where $|(\mu(a,t),a,t)|$ is the length of the corresponding binary encoding of the triple $(\mu(a,t),a,t)$ (note that $\mu(a,t)$ is a rational number).

A direct corollary to Lemma 7 and Lemma 9 is the following:

Corollary 10. Let us assume that Act is finite and each $s \in S$ is well-defined. Then \neq over $S \times S$ is semidecidable.

3 Deciding Bisimilarity Over pBPA and pBPP Processes

In this section we show that bisimilarity is decidable over pBPA and pBPP processes, which are probabilistic extensions of the well-known process classes BPA and BPP [9]. Moreover, we also show that bisimilarity over normed subclasses of pBPA and pBPP is decidable in polynomial time.

Let S = (S, Act, D) be a transition system, and let "." be a binary operator on S. For every $R \subseteq S \times S$, the symbol $\stackrel{R}{=}$ denotes the least congruence over S wrt. "." subsuming R.

Lemma 11. Let $R \subseteq S \times S$, and let Pre(R) be the least set such that $R \subseteq Pre(R)$, and if $(s,t) \in Pre(R)$ then also (su, tu), $(us, ut) \in Pre(R)$ for every $u \in S$. Then $\equiv_{Pre(R)} = \frac{R}{\equiv}$.

Now we formulate three abstract conditions which guarantee the semidecidability of \simeq over $S \times S$. As we shall see, pBPA and pBPP classes satisfy these conditions.

- 1. For every finite relation $R \subseteq S \times S$ we have that if R expands in $\stackrel{R}{\equiv}$, then $\stackrel{R}{\equiv} \subseteq \simeq$.
- 2. There is a finite relation $\mathcal{B} \subseteq S \times S$ such that $\stackrel{\mathcal{B}}{=} \simeq$ over $S \times S$ (\mathcal{B} is called a *bisimulation base*).
- 3. The definition of S is effective in the following sense: the set of states S is recursively enumerable, each state $s \in S$ is well-defined, and the problem if $s = t \cdot u$ for given $s, t, u \in S$ is semidecidable.

¹ Strictly speaking, we consider expansion in $E \cup \{(s, s) \mid s \in S\}$ because E is not an equivalence over S (which is required by Definition 2).

Lemma 12. If the three conditions above are satisfied, then \simeq over $S \times S$ is semidecidable (and thus decidable by applying Corollary 10).

Now we formally introduce pBPA and pBPP processes. Let $N = \{X, Y, ...\}$ be a countably infinite set of *constants* and $Act = \{a, b, ...\}$ a countably infinite set of actions. The elements of N^* are denoted $\alpha, \beta, ...,$ and the empty word by ε .

Let $\mu \in Disc(Act \times N^*)$ be a distribution. For each $\alpha \in N^*$, the symbol $\mu \alpha$ denotes the distribution such that $(\mu \alpha)(a, \beta \alpha) = \mu(a, \beta)$, and $(\mu \alpha)(a, \gamma) = 0$ if α is not a suffix of γ .

Definition 13. A pBPA (pBPP) system Δ is a finite set of rules of the form $X \rightarrow \mu$ where $\mu \in Disc(Act \times N^*)$ is a rational distribution with a finite support.

The sets of all constants and actions occurring in Δ are denoted $N(\Delta)$ and $Act(\Delta)$, respectively. We require that for each $X \in N(\Delta)$ there is at least one rule of the form $X \to \mu$ in Δ .

To Δ we associate the transition system $S_{\Delta} = (N(\Delta)^*, Act(\Delta), D)$ where the transitions of D are determined as follows:

$$\frac{X \to \nu \in \Delta}{X \alpha \to \nu \alpha} \ \alpha \in N(\Delta)^*$$

The elements of $N(\Delta)^*$ are called pBPA processes (of Δ).

pBPP systems and processes are defined in the same way, but the elements of $N(\Delta)^*$ are understood modulo commutativity (intuitively, this corresponds to an unsynchronized parallel composition of constants).

Observe that "ordinary", i.e., non-probabilistic BPA and BPP systems can be understood as those pBPA and pBPP where all distributions used in basic transitions are Dirac (see Section 2). Moreover, to every pBPA/pBPP system Δ we associate its *underlying* non-probabilistic BPA/BPP system Δ^{u} defined as follows: for every rule $X \to \mu \in \Delta$ we add to Δ^{u} the rules $X \xrightarrow{a} \alpha$ for each $(a, \alpha) \in supp(\mu)$. If we consider \simeq as a relation on the states of $S_{\Delta^{u}}$, we can readily confirm that \simeq is a (non-probabilistic) strong bisimulation; this follows immediately from Lemma 6. However, \simeq is generally *finer* than strong bisimilarity over the states of $S_{\Delta^{u}}$.

Definition 14. Let Δ be a pBPA or pBPP system. A given $X \in N(\Delta)$ is normed if there is some $w \in Act(\Delta)^*$ such that $X \xrightarrow{w} \varepsilon$. The norm of X, denoted n(X), is the length of the shortest such w. If $X \in N(\Delta)$ is not normed, we put $n(X) = \infty$. We say that Δ is normed if every $X \in N(\Delta)$ is normed.

Note that $n(\varepsilon) = 0$, and if we adopt the usual conventions for ∞ , then $n(\alpha\beta) = n(\alpha) + n(\beta)$. Also note that bisimilar processes must have the same norm. Transition systems generated by pBPA and pBPP systems are clearly effective in the sense of condition 3 above. Now we check that conditions 1 and 2 are also satisfied. This is where new problems (which are specific to the probabilistic setting) arise.

Lemma 15 (Condition 1). Let Δ be a pBPA or a pBPP system. Let R be a binary relation over $N(\Delta)^*$, and let E be a congruence over $N(\Delta)^*$ where $R \subseteq E$. If R expands in E, then $\stackrel{R}{=}$ expands in E.

It follows from Lemma 15 that $\stackrel{R}{\equiv} \subseteq \simeq$ whenever *R* expands in $\stackrel{R}{\equiv}$.

Corollary 16. \simeq is a congruence over processes of a given pBPA or pBPP system.

Proof. \simeq expands in $\stackrel{\simeq}{\equiv}$, hence $\stackrel{\simeq}{\equiv} \subseteq \simeq$ by Lemma 15.

It remains to check that bisimilarity over pBPA and pBPP processes can be represented by a finite base (condition 2 above).

Lemma 17 (Condition 2 for pBPP). Let Δ be a pBPP system. There is a finite relation $\mathcal{B} \subseteq N(\Delta)^* \times N(\Delta)^*$ such that $\stackrel{\mathcal{B}}{=} \simeq over N(\Delta)^* \times N(\Delta)^*$.

Proof. The proof in [9] for (non-probabilistic) BPP relies just on the fact that (non-probabilistic) bisimilarity is a congruence. Due to Corollary 16, we can use the same proof also for pBPP. \Box

In the case of pBPA, the situation is more complicated. Let $N_n \subseteq N(\Delta)$ be the set of all normed variables, and $N_u = N(\Delta) \setminus N_n$ the set of all unnormed ones.

Lemma 18. Let $X \in N(\Delta)$ and $\alpha \in N(\Delta)^*$. If $n(X) = \infty$, then $X \simeq X\alpha$.

Note that due to Lemma 18 we need only ever consider states $\alpha \in N_n^* \cup (N_n^* \times N_u)$, the others being immediately transformed into such a bisimilar state by erasing all symbols following the first infinite-norm variable.

A careful inspection of the construction for non-probabilistic BPA (as presented in [9]) reveals the following:

Proposition 19 (See [9]). Let Δ be a (non-probabilistic) BPA system. Let $\stackrel{\circ}{=} \subseteq N(\Delta)^* \times N(\Delta)^*$ be an equivalence satisfying the following properties:

- 1. if $\alpha \stackrel{*}{=} \beta$ and $\alpha \stackrel{w}{\to} \alpha'$, then there is $\beta \stackrel{w}{\to} \beta'$ such that $\alpha' \stackrel{*}{=} \beta'$ (note that it implies that $n(\alpha) = n(\beta)$);
- 2. [≜] is a congruence;
- 3. if $\alpha \gamma \stackrel{\circ}{=} \beta \gamma$ for infinitely many pairwise non-equivalent γ 's, then $\alpha \stackrel{\circ}{=} \beta$;

Then there is a finite base \mathcal{B} such that $\stackrel{\mathcal{B}}{\equiv} = \stackrel{\circ}{=} over N_n^* \cup (N_n^* \times N_u).$

So, it suffices to prove that \simeq (when considered as an equivalence over the states of the underlying BPA system Δ^{u}) satisfies the conditions 1–3 of Proposition 19. The first condition follows immediately from Lemma 6, and the second condition follows from Corollary 16. Condition 3 is proven below, together with one auxiliary result.

Lemma 20. Let α, β be processes of a pBPA system. If $\alpha \simeq \gamma \alpha$ and $\beta \simeq \gamma \beta$ for some $\gamma \neq \varepsilon$, then $\alpha \simeq \beta$.

Lemma 21. Let α, β be processes of a pBPA system. If $\alpha \gamma \simeq \beta \gamma$ for infinitely many pairwise non-bisimilar γ 's, then $\alpha \simeq \beta$.

An immediate consequence of Proposition 19, Lemma 6, Corollary 16, and Lemma 21, is the following:

Lemma 22 (Condition 2 for pBPA). Let Δ be a pBPA system. There is a finite relation $\mathcal{B} \subseteq N(\Delta)^* \times N(\Delta)^*$ such that $\stackrel{\mathcal{B}}{\equiv} \simeq over N_n^* \cup (N_n^* \times N_u)$.

Now we can formulate the first theorem of our paper:

Theorem 23. *Bisimilarity for pBPA and pBPP processes is decidable.*

3.1 Polynomial-Time Algorithms for Normed pBPA and Normed pBPP

In this subsection we show that the polynomial-time algorithms deciding (nonprobabilistic) bisimilarity over the normed subclasses of BPA and BPP processes (see [9]) can also be adapted to the probabilistic case. We concentrate just on crucial observations which underpin the functionality of these algorithms, and show that they can be reformulated and reproved in the probabilistic setting. We refer to [9] for the omitted parts.

In the probabilistic setting, the polynomial-time algorithms deciding nonprobabilistic bisimilarity over normed BPA and normed BPP processes are modified as follows: Given a normed pBPA or normed pBPP system Δ , we run the non-probabilistic algorithm on the underlying system Δ^u , where the only modification is that *the expansion is considered in the probabilistic transition system* S_{Δ} (instead of S_{Δ^u}). To see that the modified algorithm is again polynomial-time, we need to realize that the problem if a given pair of pBPA or pBPP processes expands in a polynomially computable equivalence is decidable in polynomial time. However, it is a simple consequence of Lemma 9.

Lemma 24. Let Δ be a pBPA or pBPP system, and E a polynomially computable equivalence over $N(\Delta)^*$. Let α, β be processes of Δ . It is decidable in polynomial time whether (α, β) expands in E.

The authors have carefully verified that bisimilarity has all the properties which imply the correctness of these (modified) algorithms. Some of the most important observations are listed below; roughly speaking, the original nonprobabilistic algorithms are based mainly on the unique decomposition property, which must be reestablished in the probabilistic setting.

A pBPA or pBPP process α is a *prime* iff whenever $\alpha \simeq \beta \gamma$, then either $\beta = \varepsilon$ or $\gamma = \varepsilon$ (note that $\alpha \in N$).

Lemma 25. Let α, β, γ be processes of a normed pBPA system. Then $\alpha \gamma \simeq \beta \gamma$ implies $\alpha \simeq \beta$.
Theorem 26. Every normed pBPA process α decomposes uniquely (up to bisimilarity) into prime components.

Proof. We can use the same proof as in [9]. It relies on Lemma 25, Corollary 16, and Lemma 6. \Box

Theorem 27. Every normed pBPP process decomposes uniquely (up to bisimilarity) into prime components.

Proof. As in [9]. It relies on Lemma 6.

Now we have all the "tools" required for adapting the observations about non-probabilistic normed BPA/BPP to the probabilistic setting which altogether imply the following:

Theorem 28. Bisimilarity is decidable for normed pBPA and normed pBPP processes in polynomial time.

4 Deciding Bisimilarity Between pPDA and pFS Processes

Definition 29. A probabilistic pushdown automaton (pPDA) is a tuple $\Delta = (Q, \Gamma, Act, \delta)$ where Q is a finite set of control states, Γ is a finite stack alphabet, Act is a finite set of actions, and $\delta : (Q \times \Gamma) \rightarrow 2^{\text{Disc}(Act \times (Q \times \Gamma^*))}$ is a transition function such that the set $\delta(p, X)$ is finite and each $\mu \in \delta(p, X)$ is a rational distribution with a finite support for all $p \in Q$ and $X \in \Gamma$.

We write $p\alpha$ instead of (p, α) and $pA \rightarrow \mu$ instead of $\mu \in \delta(p, A)$. Let $\nu \in Disc(Act \times (Q \times \Gamma^*))$ be a distribution. For each $\beta \in \Gamma^*$, the symbol $\nu\beta$ denotes the distribution such that $(\nu\beta)(a, p\alpha\beta) = \nu(a, p\alpha)$, and $(\nu\beta)(a, p\gamma) = 0$ if β is not a suffix of γ . Each pPDA Δ induces a unique transition system S_{Δ} where $Q \times \Gamma^*$ is the set of states, Act is the set of actions, and transitions are given by the following rule:

$$\frac{pX \to \nu \in \delta}{pX\beta \to \nu\beta} \ \beta \in \Gamma^*$$

The states of S_{Δ} are called pPDA processes of Δ , or just pPDA processes if Δ is not significant.

Our aim is to show that \simeq between pPDA processes and finite-state processes is decidable in exponential time. For this purpose we adapt the results of [19], where a generic framework for deciding various behavioral equivalences between PDA and finite-state processes is developed. In this framework, the generic part of the problem (applicable to every behavioral equivalence which is a right PDA congruence in the sense of Definition 31) is clearly separated from the equivalence-specific part that must be supplied for each behavioral equivalence individually. The method works also in the probabilistic setting, but

the application part would be unnecessarily complicated if we used the original scheme proposed in [19]. Therefore, we first develop the generic part of the method into a more "algebraic" form, and then apply the new variant to probabilistic bisimilarity. The introduced modification is generic and works also for other (non-probabilistic) behavioral equivalences.

For the rest of this section, we fix a pPDA $\Delta = (Q, \Gamma, Act, \delta)$ of size m and a finite-state system $\mathcal{S} = (F, Act, D)$ of size n (the size of a given $\mu \in Disc(Act \times (Q \times \Gamma^*))$ is defined similarly as in Lemma 9). In our complexity estimations we also use the parameter $z = |F|^{|Q|}$.

We start by recalling some notions and results of [19]. To simplify our notation, we introduce all notions directly in the probabilistic setting. We denote $F_{\perp} = F \cup \{\perp\}$, where $\perp \notin F$ stands for "undefined".

Definition 30. For every process $p\alpha$ of Δ we define the set $M_{p\alpha} = \{q \in Q \mid p\alpha \rightarrow^* q\epsilon\}$. A function $\mathcal{F} : Q \rightarrow F_{\perp}$ is compatible with $p\alpha$ iff $\mathcal{F}(q) \neq \perp$ for every $q \in M_{p\alpha}$. The class of all functions that are compatible with $p\alpha$ is denoted $Comp(p\alpha)$.

For every process $p\alpha$ of Δ and every $\mathcal{F} \in Comp(p\alpha)$ we define the process $p\alpha \mathcal{F}$ whose transitions are determined by the following rules:

$$\frac{p\alpha \to \mu}{p\alpha \mathcal{F} \to \mu \mathcal{F}} \mathcal{F} \in Comp(p\alpha) \qquad \qquad \frac{\mathcal{F}(p) \to \mu}{p\mathcal{F} \to \mu_{\mathcal{F}}} \mathcal{F} \in Comp(p\varepsilon)$$

Here $\mu \mathcal{F}$ is a distribution which returns a non-zero value only for pairs of the form $(a, q\beta \mathcal{F})$, where $(\mu \mathcal{F})(a, q\beta \mathcal{F}) = \mu(a, q\beta)$, and $\mu_{\mathcal{F}}$ is a distribution which returns a non-zero value only for pairs of the form $(a, p\mathcal{F}[s/p])$, where $\mu(a, p\mathcal{F}[s/p]) = \mu(a, s)$. Here $\mathcal{F}[s/p] : Q \to F_{\perp}$ is the function which returns the same result as \mathcal{F} for every argument except for p where $\mathcal{F}[s/p](p) = s$. In other words, $p\alpha\mathcal{F}$ behaves like $p\alpha$ until the point when the stack is emptied and a configuration of the form $q\varepsilon$ is entered; from that point on, $p\alpha\mathcal{F}$ behaves like $\mathcal{F}(q)$. Note that if $\mathcal{F} \in Comp(p\alpha)$ and $p\alpha \to^* q\beta$, then $\mathcal{F} \in Comp(q\beta)$. We also put $Stack(\Delta, F) = \Gamma^* \cup \{\alpha\mathcal{F} \mid \alpha \in \Gamma^*, \mathcal{F} \in (F_{\perp})^Q\}$, and $\mathcal{P}(\Delta, F) = \{p\alpha \mid p \in Q, \alpha \in \Gamma^*\} \cup \{p\alpha\mathcal{F} \mid p \in Q, \alpha \in \Gamma^*, \mathcal{F} \in Comp(p\alpha)\}$.

Definition 31. We say that an equivalence E over $\mathcal{P}(\Delta, F) \cup F$ is a right pPDA congruence (for Δ and S) iff the following conditions are satisfied:

- For every process $p\alpha$ of Δ and all $\varphi, \psi \in Stack(\Delta, F)$ we have that if $(q\varphi, q\psi) \in E$ for each $q \in M_{p\alpha}$, then also $(p\alpha\varphi, p\alpha\psi) \in E$. - $(p\mathcal{F}, \mathcal{F}(p)) \in E$ for every $p\mathcal{F} \in \mathcal{P}(\Delta, F)$.

Let *R* be a binary relation over $\mathcal{P}(\Delta, F) \cup F$. The least right pPDA congruence over $\mathcal{P}(\Delta, F) \cup F$ subsuming *R* is denoted $\stackrel{R}{\equiv}_r$. Further, Rpre(R) denotes the least relation over $\mathcal{P}(\Delta, F) \cup F$ subsuming *R* satisfying the following condition: For every process $p\alpha$ of Δ and all $\varphi, \psi \in Stack(\Delta, F)$ we have that if $(q\varphi, q\psi) \in$ Rpre(R) for each $q \in M_{p\alpha}$, then also $(p\alpha\varphi, p\alpha\psi) \in Rpre(R)$. In general, $\equiv_{Rpre(R)}$ is a *proper* subset of $\stackrel{R}{\equiv}_r$; the relationship between Rpre(R) and $\stackrel{R}{\equiv}_r$ is revealed in the following lemma: **Lemma 32.** Let R be a binary relation over $\mathcal{P}(\Delta, F) \cup F$. For every $i \in \mathbb{N}_0$ we define a binary relation R^i over $\mathcal{P}(\Delta, F) \cup F$ inductively as follows: $R^0 = R$, and $R^{i+1} = \equiv_{Rpre(R^i)}$. Then $\stackrel{R}{\equiv}_r = \bigcup_{i \in \mathbb{N}_0} R^i$.

For the rest of this section, let us fix a right pPDA congruence $\stackrel{\circ}{=}$ over $\mathcal{P}(\Delta, F) \cup F$ which is decidable for finite-state processes and satisfies the following transfer property: if $s \stackrel{\circ}{=} t$ and $s \rightarrow^* s'$, then there exists t' such that $t \rightarrow^* t'$ and $s' \stackrel{\circ}{=} t'$. The following definitions are also borrowed from [19].

Definition 33. Let $\varphi \in Stack(\Delta, F)$ and $\mathcal{F} : Q \to F_{\perp}$. We write $\varphi \stackrel{*}{=} \mathcal{F}$ iff for all $p \in Q$ we have that if $\mathcal{F}(p) \neq \perp$, then $p\varphi \stackrel{*}{=} \mathcal{F}(p)$.

Further, for every relation $K \subseteq Stack(\Delta, F) \times (F_{\perp})^{Q}$ we define the set I(K) of K-instances as follows: $I(K) = \{(p\varphi, \mathcal{F}(p)) \mid (\varphi, \mathcal{F}) \in K, \mathcal{F}(p) \neq \bot\}.$

Definition 34. Let $K = \{(\varepsilon, \mathcal{F}) | \varepsilon \stackrel{\circ}{=} \mathcal{F}\} \cup \{(\mathcal{G}, \mathcal{F}) | \mathcal{G} \stackrel{\circ}{=} \mathcal{F}\} \cup K'$ where $K' \subseteq \Gamma \times (F_{\perp})^{Q} \cup ((\Gamma \times (F_{\perp})^{Q}) \times (F_{\perp})^{Q})$. (That is, K' consists of (some) pairs of the form (X, \mathcal{F}) and $(X\mathcal{G}, \mathcal{F})$). We say that K is well-formed iff K satisfies the following conditions:

- if $(X\mathcal{G}, \mathcal{F}) \in K$ and $\mathcal{F}(p) \neq \bot$, then $\mathcal{G} \in Comp(pX)$;
- if $(X, \mathcal{F}) \in K$ (or $(X\mathcal{G}, \mathcal{F}) \in K$) and $(\mathcal{F}, \mathcal{H}) \in K$, then also $(X, \mathcal{H}) \in K$ (or $(X\mathcal{G}, \mathcal{H}) \in K$, resp.).

It is clear that there are only finitely many well-formed sets, and that there exists the greatest well-formed set G whose size is $\mathcal{O}(|\Gamma| \cdot |F|^{2 \cdot |Q|})$. Observe that G is effectively constructible because $\stackrel{\circ}{=}$ is decidable for finite-state processes.

Intuitively, well-formed sets are finite representations of certain infinite relations between processes of $\mathcal{P}(\Delta, F)$ and F, which are "generated" from wellformed sets using the rules introduced in our next definition:

Definition 35. Let K be a well-formed set. The closure of K, denoted Cl(K), is the least set L satisfying the following conditions:

(1) $K \subseteq L$;

(2) if $(\alpha \mathcal{G}, \mathcal{F}) \in L$, $(\varepsilon, \mathcal{G}) \in K$, and $\alpha \neq \varepsilon$, then $(\alpha, \mathcal{F}) \in L$;

- (3) if $(\alpha \mathcal{G}, \mathcal{F}) \in L$, $(\mathcal{H}, \mathcal{G}) \in K$, and $\alpha \neq \varepsilon$, then $(\alpha \mathcal{H}, \mathcal{F}) \in L$;
- (4) if $(\alpha \mathcal{G}, \mathcal{F}) \in L$, $(X, \mathcal{G}) \in K$, and $\alpha \neq \varepsilon$, then $(\alpha X, \mathcal{F}) \in L$;
- (5) if $(\alpha \mathcal{G}, \mathcal{F}) \in L$, $(X\mathcal{H}, \mathcal{G}) \in K$, and $\alpha \neq \varepsilon$, then $(\alpha X\mathcal{H}, \mathcal{F}) \in L$.

Further, we define Gen(K) = I(Cl(K)).

Observe that *Cl* and *Gen* are monotonic and that $Gen(K) \subseteq \mathcal{P}(\Delta, F) \times F$ for every well-formed set *K*.

An important property of *Gen* is that it generates only "congruent pairs" as stated in the following lemma.

Lemma 36. Let K be a well-formed set. Then $Gen(K) \subseteq \stackrel{I(K)}{\equiv_r}$.

The following well-formed set is especially important.

Definition 37. The base \mathcal{B} is defined as follows: $\mathcal{B} = \{(\varepsilon, \mathcal{F}) \mid \varepsilon \triangleq \mathcal{F}\} \cup \{(\mathcal{G}, \mathcal{F}) \mid \mathcal{G} \triangleq \mathcal{F}\} \cup \{(X, \mathcal{F}) \mid X \triangleq \mathcal{F}\} \cup \{(X\mathcal{G}, \mathcal{F}) \mid X\mathcal{G} \triangleq \mathcal{F}\}.$

The importance of \mathcal{B} is clarified in the next lemma.

Lemma 38 (see [19]). $Gen(\mathcal{B})$ coincides with $\stackrel{\circ}{=} over \mathcal{P}(\Delta, F) \times F$.

Let (\mathcal{W}, \subseteq) be the complete lattice of all well-formed sets, and let $Exp : \mathcal{W} \to \mathcal{W}$ be a function satisfying the four conditions listed below:

- 1. $Exp(\mathcal{B}) = \mathcal{B}$.
- 2. Exp is monotonic, i.e. $K \subseteq L$ implies $Exp(K) \subseteq Exp(L)$.
- 3. If K = Exp(K), then $Gen(K) \subseteq \stackrel{\circ}{=}$.
- 4. The membership to Exp(K) is decidable.

According to condition 1, the base \mathcal{B} is a fixed-point of *Exp*. We prove that \mathcal{B} is the greatest fixed-point of *Exp*. Suppose that K = Exp(K) for some well-formed set K. By definition of Gen(K) and condition 3 we have that $I(K) \subseteq I(Cl(K)) = Gen(K) \subseteq \stackrel{\circ}{=}$. Since for each $(\varphi, \mathcal{F}) \in K$ we have that $\mathcal{F}(p) \neq \bot$ implies $p\varphi \stackrel{\circ}{=} \mathcal{F}(p)$, we can conclude that $(\varphi, \mathcal{F}) \in \mathcal{B}$.

Hence, \mathcal{B} can be computed by a simple algorithm which iterates *Exp* on *G* until a fixed-point is found. These conditions are formulated in the same way as in [19] except for condition 3 which is slightly different. As we shall see, with the help of the new "algebraic" observations presented above, condition 3 can be checked in a relatively simple way. This is the main difference from the original method presented in [19].

Similarly as in [19], we use finite multi-automata to represent certain infinite subsets of $\mathcal{P}(\Delta, F)$.

Definition 39. A multi-automaton is a tuple $\mathcal{M} = (S, \Sigma, \gamma, Acc)$ where

- S is a finite set of states such that $Q \subseteq S$ (i.e, the control states of Δ are among the states of \mathcal{M});
- $-\Sigma = \Gamma \cup \{\mathcal{F} \mid \mathcal{F} : Q \to F_{\perp}\} \text{ is the input alphabet (the alphabet has a special symbol for each } \mathcal{F} : Q \to F_{\perp});$
- $-\gamma \subseteq S \times \Sigma \times S$ is a transition relation;
- $Acc \subseteq S$ is a set of accepting states.

Every multi-automaton $\mathcal M$ determines a unique set

$$\mathcal{L}(\mathcal{M}) = \{ pw \mid p \in Q, w \in \Sigma^*, \gamma(p,w) \cap Acc
eq \emptyset \}$$

The following tool will be useful for deciding the membership to Exp(K).

Lemma 40. Let K be a well-formed set. The relation $R = (\equiv_{Gen(K)} \cap (F \times F))$ is computable in time polynomial in m, n, z. Moreover, for each equivalence class $C \in F/R$ there is a multiautomaton $\mathcal{M}_{K,C}$ accepting the set $C' \subseteq \mathcal{P}(\Delta, F)$ where $C \cup C' \in (\mathcal{P}(\Delta, F) \cup F) / \equiv_{Gen(K)}$. The automaton $\mathcal{M}_{K,C}$ is constructible in time polynomial in m, n, z.

4.1 Deciding \simeq Between pPDA and Finite-State Processes

We apply the abstract framework presented in the previous section. That is, we show that \simeq is a right pPDA congruence and define an appropriate function *Exp* satisfying the four conditions given earlier. We start with an auxiliary result.

Lemma 41. Let *R* be a binary relation over $\mathcal{P}(\Delta, F) \cup F$. If *R* expands in $\stackrel{R}{\equiv}_{r}$ then $\stackrel{R}{\equiv}_{r} \subseteq \simeq$.

The next lemma follows immediately from Lemma 41.

Lemma 42. \simeq is a right pPDA congruence.

Definition 43. Given a well-formed set K, the set Exp(K) consists of all pairs $(\varphi, \mathcal{F}) \in K$ such that for each $p \in Q$ we have that if $\mathcal{F}(p) \neq \bot$, then $(p\varphi, \mathcal{F}(p))$ expands in $\equiv_{Gen(K)}$.

Now we verify the four conditions that must be satisfied by *Exp*. The first condition follows easily from the fact that $Gen(\mathcal{B})$ coincides with \simeq over $\mathcal{P}(\Delta, F) \times F$, because if $(p\varphi, \mathcal{F}(p)) \in I(\mathcal{B})$, then $\simeq = \equiv_{Gen(\mathcal{B})} \text{ over } succ(p\varphi) \cup succ(\mathcal{F}(p))$. The second condition is obvious.

Lemma 44. Exp(K) = K implies $\equiv_{Gen(K)} \subseteq \simeq$.

Proof. Exp(K) = K implies that each pair of I(K) expands in $\equiv_{Gen(K)}$. But then each pair of I(K) expands in $\stackrel{I(K)}{\equiv_r}$ by Lemma 3 and Lemma 36. Thus, $\equiv_{Gen(K)} \subseteq \stackrel{I(K)}{\equiv_r} \subseteq \simeq$ by Lemma 41.

Lemma 45. Exp(K) is computable in time polynomial in m, n, z.

Proof. Let $(p\alpha, \mathcal{F}(p)) \in I(K)$ and $U = succ(p\alpha) \cup succ(\mathcal{F}(p))$. It follows immediately from Lemma 40 that the equivalence relation $\equiv_{Gen(K)} \cap (U \times U)$ can be computed in time polynomial in m, n, z. The claim then follows from Lemma 9.

Now we can formulate our next theorem.

Theorem 46. Probabilistic bisimilarity between pPDA and finite-state processes is decidable in time which is polynomial in m, n, z. That is, the problem is decidable in exponential time for general pPDA, and in polynomial time for every subclass of pPDA where the number of control states is bounded by some constant (in particular, this applies to pBPA).

Proof. Let $p\alpha$ be a pPDA process and f a finite-state process. We can assume (w.l.o.g.) that $\alpha = X$ for some $X \in \Gamma$. The algorithm computes the base \mathcal{B} by first computing the greatest well-formed relation G and then iterating *Exp* until a fixed-point is found. Then, it suffices to find out if there is a pair $(X, \mathcal{F}) \in \mathcal{B}$ such that $\mathcal{F}(p) = f$. Note that this takes time polynomial in m, n, z, because

- G is computable in time polynomial in m, n, z. This is because the size of G is $\mathcal{O}(|\Gamma| \cdot |F|^{2 \cdot |Q|})$ and \simeq over finite-state systems is decidable in polynomial time [10].
- Exp is computable in time polynomial in m, n, z due to Lemma 45.
- The algorithm needs at most |G|, i.e., $\mathcal{O}(|\Gamma| \cdot |F|^{2 \cdot |Q|})$ iterations to reach a fixed-point.

5 Conclusions

The results presented in this paper show that various forms of probabilistic bisimilarity are decidable over certain classes of infinite-state systems. In particular, this paper advocates the use of algebraic methods which were originally developed for non-probabilistic systems. These methods turn out to be surprisingly robust and can be applied also in the probabilistic setting.

An obvious question is whether the decidability/tractability results for other non-probabilistic infinite-state models can be extended to the probabilistic case. We conjecture that the answer is positive in many cases, and we hope that the results presented in this paper provide some hints and guidelines on how to achieve that. Another interesting question is whether we could do better than in the non-probabilistic case. In particular, undecidability results and lower complexity bounds do not carry over to fully probabilistic variants of infinite-state models (fully probabilistic systems are probabilistic systems where each state *s* has at most one out-going transition $s \rightarrow \mu$). It is still possible that methods specifically tailored to fully probabilistic models might produce better results than their non-probabilistic counterparts. This also applies to probabilistic variants of other behavioural equivalences, such as trace or simulation equivalence.

References

- P.A. Abdulla, C. Baier, S.P. Iyer, and B. Jonsson. Reasoning about probabilistic channel systems. In *Proceedings of CONCUR 2000*, volume 1877 of *LNCS*, pages 320–330. Springer, 2000.
- [2] P.A. Abdulla and A. Rabinovich. Verification of probabilistic systems with faulty communication. In *Proceedings of FoSSaCS 2003*, volume 2620 of *LNCS*, pages 39–53. Springer, 2003.
- [3] A. Aziz, V. Singhal, F. Balarin, R. Brayton, and A. Sangiovanni-Vincentelli. It usually works: The temporal logic of stochastic systems. In *Proceedings of CAV'95*, volume 939 of *LNCS*, pages 155–165. Springer, 1995.
- [4] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. On the consistency of Koomen's fair abstraction rule. TCS, 51(1):129–176, 1987.
- [5] C. Baier and B. Engelen. Establishing qualitative properties for probabilistic lossy channel systems: an algorithmic approach. In *Proceedings of 5th International AMAST Workshop on Real-Time and Probabilistic Systems (ARTS'99)*, volume 1601 of *LNCS*, pages 34–52. Springer, 1999.

- [6] C. Baier, H. Hermanns, and J. Katoen. Probabilistic weak simulation is decidable in polynomial time. *Information Processing Letters*, 89(3): 123–130, 2004.
- [7] A. Bianco and L. de Alfaro. Model checking of probabalistic and nondeterministic systems. In *Proceedings of FST&TCS'95*, volume 1026 of *LNCS*, pages 499–513. Springer, 1995.
- [8] T. Brázdil, A. Kučera, and O.Stražovský. Deciding probabilistic bisimilarity over infinite-state probabilistic systems. Technical report FIMU-RS-2004-06, Faculty of Informatics, Masaryk University, 2004.
- [9] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. *Handbook of Process Algebra*, pages 545–623, 1999.
- [10] S. Cattani and R. Segala. Decision algorithms for probabilistic bisimulation. In *Proceedings of CONCUR 2002*, volume 2421 of *LNCS*, pages 371–385. Springer, 2002.
- [11] C. Courcoubetis and M. Yannakakis. Verifying temporal properties of finite-state probabilistic programs. In *Proceedings of 29th Annual Symposium on Foundations* of Computer Science, pages 338–345. IEEE, 1988.
- [12] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. JACM, 42(4):857–907, 1995.
- [13] L. de Alfaro, M.Z. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In *Proceedings of TACAS 2000*, volume 1785 of *LNCS*, pages 395–410. Springer, 2000.
- [14] J. Esparza, A. Kučera, and R. Mayr. Model-checking probabilistic pushdown automata. In *Proceedings of LICS 2004*. IEEE, 2004. To appear.
- [15] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:512–535, 1994.
- [16] M. Huth and M.Z. Kwiatkowska. Quantitative analysis and model checking. In Proceedings of LICS'97, pages 111–122. IEEE, 1997.
- [17] S.P. Iyer and M. Narasimha. Probabilistic lossy channel systems. In *Proceedings of TAPSOFT'97*, volume 1214 of *LNCS*, pages 667–681. Springer, 1997.
- [18] B. Jonsson, W. Yi, and K.G. Larsen. Probabilistic extensions of process algebras. *Handbook of Process Algebra*, pages 685–710, 1999.
- [19] A. Kučera and R. Mayr. A generic framework for checking semantic equivalences between pushdown automata and finite-state automata. In *Proceedings of IFIP TCS*'2004. Kluwer, 2004. To appear.
- [20] M.Z. Kwiatkowska. Model checking for probability and time: from theory to practice. In *Proceedings of LICS 2003*, pages 351–360. IEEE, 2003.
- [21] K. Larsen and A. Skou. Bisimulation through probabilistic testing. *I&C*, 94(1):1–28, 1991.
- [22] A. Rabinovich. Quantitative analysis of probabilistic lossy channel systems. In *Proceedings of ICALP 2003*, volume 2719 of *LNCS*, pages 1008–1021. Springer, 2003.
- [23] R. Segala and N.A. Lynch. Probabilistic simulations for probabilistic processes. NJC, 2(2):250–273, 1995.

µABC: A Minimal Aspect Calculus

Glenn Bruns¹, Radha Jagadeesan^{2*}, Alan Jeffrey^{2**}, and James Riely^{2***}

¹ Bell Labs, Lucent Technologies ² DePaul University

Abstract. Aspect-oriented programming is emerging as a powerful tool for system design and development. In this paper, we study aspects as primitive computational entities on par with objects, functions and horn-clauses. To this end, we introduce μABC , a name-based calculus, that incorporates aspects as primitive. In contrast to earlier work on aspects in the context of object-oriented and functional programming, the only computational entities in μABC are aspects. We establish a compositional translations into μABC from a functional language with aspects and higher-order functions. Further, we delineate the features required to support an aspect-oriented style by presenting a translation of μABC into an extended π -calculus.

1 Introduction

Aspects [7,21,28,23,22,3] have emerged as a powerful tool in the design and development of systems (e.g., see [4]). To explain the interest in aspects, we begin with a short example inspired by tutorials of AspectJ [1]. Suppose class *L* realizes a useful library, and that we want to obtain timing information about a method foo () of *L*. With aspects this can be done by writing *advice* specifying that, whenever foo is called, the current time should be logged, foo should be executed, and then the current time should again be logged. It is indicative of the power of the aspect framework that:

- the profiling code is localized in the advice,
- the library source code is left untouched, and
- the responsibility for profiling all foo() calls resides with the compiler and/or runtime environment.

The second and third items ensure that, in developing the library, one need not worry about advice that may be written in the future. In [13] this notion is called *obliviousness*. However, in writing the logging advice, one must identify the pieces of code that need to be logged. In [13] this notion is called *quantification*. These ideas are quite general and are independent of programming language paradigm.

The execution of such an aspect-program can intuitively be seen in a reactive framework as follows. View method invocations (in this case the $f\infty()$ invocations) as

^{*} Supported by NSF grant #0244901.

^{**} Supported by NSF grant #0208549.

^{***} Supported by NSF grant #0347542.

events. View advice code (in this case the logging advice) as running in parallel with the other source code and responding to occurrences of events (corresponding to method calls). This view of execution is general enough to accommodate dynamic arrival of new advice by treating it as dynamically created parallel components. In the special case that all advice is static, the implicit parallel composition of advice can be compiled away — in aspect-based languages, this compile-time process called *weaving*. Informally, the weaving algorithm replaces each call to foo() with a call to the advice code, thus altering the client code and leaving the library untouched.

Aspect-oriented extensions have been developed for object-oriented [21,28], imperative [20], and functional languages [30,31]. Furthermore, a diverse collection of examples show the utility of aspects. These range from the treatment of inheritance anomalies in concurrent object-oriented programming (eg. see [25] for a survey of such problems, and [24] for an aspect-based approach) to the design of flexible mechanisms for access control in security applications [5], Recent performance evaluations of aspect languages [12] suggest that a combination of programming and compiler efforts suffices to manage any performance penalties.

Much recent work on aspects is aimed at improving aspect-oriented language design and providing solutions to the challenge of reasoning about aspect-oriented programs. For example, there is work on adding aspects to existing language paradigms [30,31], on finding a parametric way to describe a wide range of aspect languages [10], on finding abstraction principles [11], on type systems [18], and on checking the correctness of compiling techniques using operational models [19] or denotational models [32]. A strategy in much of this work is to develop an calculus that provides a manageable setting in which to study the issues. Similarly to the way that aspect languages have been designed by adding aspects to an existing programming paradigm, these calculi generally extend a base calculus with a notion of aspect. For example, [19] is based on an untyped class-based calculus, [10] is based on the object calculus [2], and [31] is based on the simply-typed lambda calculus.

If one wishes to study aspects in the context of existing programming languages, then calculi of this style are quite appropriate. However, another role for an aspect calculus is to identify the essential nature of aspects and understand their relationship to other basic computational primitives. We follow the approach of the theory of concurrency — concurrency is not built on top of sequentiality because that would certainly make concurrency more complex rather than sequentiality. Rather, concurrency theory studies interaction and concurrency as primitive concepts and sequentiality emerges as a special case of concurrency.

Along these lines, we aim here to establish aspects as primitive computational entities on par with objects, functions, and horn clauses; separate from their integration into existing programming paradigms. To this end we have created a minimal aspect calculus called μABC .

We present μABC as a sequential deterministic calculus, with all concurrency being implicit. The primitive entities of μABC are names, in the style of the pi-calculus [26] and the join calculus [15]. It differs in the choice of the communication paradigm in two ways: firstly, messages are broadcast (somewhat in the style of CSP [16]) to all

receivers; secondly, the joins of the join-calculus are generalized to permit receiver code (ie. advice) to be conditional on second-order predicates over messages.

We show that functions and objects can be realized using μABC , demonstrating that aspects are an expressive primitive. Interestingly, μABC achieves expressiveness without explicit use of concurrency, providing an analysis that differs from those familiar to the concurrency community. This is not to say that aspects are incompatible with concurrency. The addition of explicit concurrency does not alter the basic development of μABC — we eschew explicit concurrency in μABC in this extended abstract to make the presentation manageable to a reader unfamiliar with aspects.

Organization. The rest of the paper is organized as follows. We begin with an informal introduction to the techniques and results of the paper. The key technical ideas are developed in the rest of the paper. Section 2 describes the syntax and dynamic semantics of μ ABC. The two following sections describe encodings of the lambda-calculus, both with and without aspects. Finally, we describe the translation of μ ABC into a variant of the polyadic pi-calculus. In this extended abstract, we elide all proofs.

2 Minimal Aspect-Based Calculus

Aspect-oriented languages add *advice* and *pointcuts* on top of events occurring in an underlying model of computation. For example, in an imperative model, the events might be procedure calls or expression evaluations. The pointcut language provides a logic for describing events or event sequences. Here we restrict our attention to single events, leaving the furtile ground of temporal pointcuts (such as AspectJ's cflow) for future work.

Advice associates a pointcut with executable code. When an event specified by the pointcut occurs, the advice "fires", intercepting the underlying event. Execution of the event itself is replaced with execution of the advice body. Advice may optionally *proceed* to execute the underlying event at any point during execution of the advice body. If many pieces of advice fire on the same event, then *advice ordering* indicates which piece of advice will be executed; in this case, a proceed will cause execution of the next piece of advice.

In μ ABC, computational events are messages sent from a source to a target. The source, message, and target are specified as names, represented as lower-case letters. An example message command is the following:

$let x = p \rightarrow q: m$

The four names in the command have different purposes, so we develop some distinguishing terminology. The source, p, and the target, q, are *roles*; m is a *message*; x is a *variable*, which binds the value returned by the message. Messages include both advice a, \ldots, e and labels f, \ldots, ℓ . Commands may specify a sequence of messages. This is useful for modeling both traditional method calls, $p \rightarrow q: \ell$, and advice sequences, $p \rightarrow q: a, b$. For compatibility with declaration order, we read message sequences right to left; in the command $p \rightarrow q: m, n$, message n is processed before m. The only other computational commands in μ ABC are return statements, which terminate all command sequences; for example, "return v" returns name v.

Finally, the calculus includes commands for declaring roles and advice. An advice declaration binds an advice name and specifies a pointcut and advice body. For example, the following advice *a* causes any message *k* sent from *p* to *q* to be redirected as a message ℓ , sent from *p* to *r*. This is an "extreme" form of delegation. Messages to *q* are delegated to *r* before *q* even receives them.

advice $a[p \rightarrow q:k] = \text{let } x = p \rightarrow r: \ell$; return x

The term between brackets is a *pointcut* indicating that message k should be intercepted when sent from p to q. The body of the advice is given after the first equality symbol.

The pointcut language allows for quantification over names. For example, the following variation captures every k-message sent to q, regardless of the sender. The advice resends the message to q, renaming it to ℓ ; the sending role is unchanged.

advice $a[\exists z \, . \, z \rightarrow q \, : \, k] = \sigma y$. let $x = y \rightarrow q \, : \, \ell$; return x

Here, z binds the source of the message in the pointcut, and y binds the source of the message in the body of the advice. The binder σ is mnemonic for "source". One may also quantify over the target of a message; the corresponding binder is τ , for "target". The following code converts every *k*-message into a *l*-message with the same source and target:

advice $a[\exists z_s . \exists z_t . z_s \rightarrow z_t : k] = \sigma y_s . \tau y_t . \text{let } x = y_s \rightarrow y_t : \ell; \text{return } x$

In all the examples given so far, the advice causes all other code associated with the event to be ignored. If we wish to allow many pieces of advice to trigger on a single event, then we must encode the ability to "proceed". The proceed binder, π , captures the names of any other advice triggered by a pointcut. The following code captures *k*-messages and executes other advice *after* redirecting the message to to ℓ .

advice $a[\exists z_s . \exists z_t . z_s \rightarrow z_t : k] = \sigma y_s . \tau y_t . \pi b . let x = y_s \rightarrow y_t : b, \ell$; return x

Reversing the order of *b* and ℓ , " $y_s \rightarrow y_t : \ell, b$ ", causes other advice to execute *before* redirecting the message. In this case, the ℓ -message will only be sent if the other advice uses its proceed binder. In general, *b* will be replaced with a sequence of messages when the advice fires. If there is no other associated advice, then *b* will be replaced with the empty sequence, in which case " $y_s \rightarrow y_t : \ell, b$ " and " $y_s \rightarrow y_t : b, \ell$ " execute identically.

 μ ABC allows bounded quantification in pointcuts. As motivation, consider a classbased language with advice, such as AspectJ. In such a language, one may specify pointcuts based on class; all objects inhabiting the class will cause the pointcut to fire. Both objects and classes are encoded in μ ABC as roles. In this case, a pointcut must specify a subset of roles as the source or target of a message. We achieve this by associating names with a partial order. The following declaration captures any k-message sent to q from a sub-name of t.

advice
$$a[\exists z \leq t \, : \, z \rightarrow q \, : \, k] = \sigma y$$
. let $x = y \rightarrow q \, : \, \ell$; return x

The partial order is established through role declarations, such as the following, which declares p to be a sub-name of q.

role p < q

In examples throughout the paper, the reserved name top is the largest name with respect to this ordering. We therefore abbreviate "role p < top" as "role p."

The role hierarchy is used extensively in the encoding of the class-based language given in the full version of the paper [9].

Dynamics. Consider the following sequence of declarations, \vec{D} .

role p; role q; role r; advice $a[p \rightarrow q:k] = \sigma y_s \cdot \tau y_t \cdot \pi b \cdot (\text{let } z = y_t \rightarrow r:b; \text{return } y_s);$

Consider the execution of the following program using \vec{D} .

\vec{D} ; let $x = p \rightarrow q$: j,k; return x

Messages are processed using two rules which differentiate the type of the leading name in the message list, in this case k. To distinguish these two forms of reduction, we impose a syntactic distinction between advice and other names. Advice is named only to simplify the semantics. The syntactic distinction makes it so that advice cannot fire based on the execution of other advice. The *advice lookup* rule replaces the leading label (or role) in a message list with the advice names that the label triggers. So k is replaced with a.

\vec{D} ; let $x = p \rightarrow q$: j, a; return x

The *advice invocation* rule replaces a message command with the appropriately instantiated body of the triggered advice. Further reducing the program by this rule, we obtain:

\vec{D} ; let $z = p \rightarrow r : j$; return p

The return variable has changed as the result of a double substitution. In the process of inserting the advice body, occurrences of the let variable x are replaced with the return value of the advice body. In this case, the return value, y_s , is itself replaced with the name of the source of the message, p.

2.1 Syntax and Semantics of µABC

Syntax. For any grammar *X*, define the grammars of lists as:

$$\vec{X} ::= X_1, \dots, X_n$$
 Comma-separated lists
 $\vec{X}; ::= X_1; \dots, X_n;$ Semicolon-terminated lists

Write $\boldsymbol{\varepsilon}$ for the empty list.

We assume a set of *names*, ranged over by m, n. We further assume that names are partitioned into two disjoint and distinguishable sets. Advice names are associated

with pointcuts and advice. Roles are use to name objects in the system as well message labels.

f,\ldots,ℓ,p,\ldots,z	Role (or Label)		
a,, e	Advice name		

Names may be roles or advice names.

 $m, n ::= \ell \mid a$ Name (or Message)

The grammar for μABC programs is as follows. We discuss point cuts, ϕ , below.

P,Q,R :=	Program
return v	Return
$\det x = p \to q : \vec{m}; P$	Message Send
role $p < q; \tilde{P}$	Role
advice $a[\phi] = \sigma x \cdot \tau y \cdot \pi b \cdot Q; P$	Advice

Let *D* and *E* range over *declarations*, which may be either role delcarations "role p < q" or advice declarations "advice $a[\phi] = \sigma x \cdot \tau y \cdot \pi b \cdot Q$ ". Let *B* and *C* range over *commands*, which may be declarations "*D*" or message sends "let $x = p \rightarrow q : \vec{m}$ ". Note that all programs have the form \vec{B} ; return *v*.

- The command "let $x = p \rightarrow q$: \vec{m} ; P" binds x in P. Execution causes messages \vec{m} to be sent from p to q, binding the return value of last executed message to x.
- The declaration "role p < q; P" binds p in P. It declares p as a subrole of q.
- The declaration "advice $a[\phi] = \sigma x \cdot \tau y \cdot \pi b \cdot Q; P$ " binds a in Q and P; in addition, x, y and z are bound in Q. It declares a to be an association between pointcut ϕ and advice body $\sigma x \cdot \tau y \cdot \pi b \cdot Q$.

Omitted binders in an advice declaration are assumed to be fresh, for example:

advice
$$[\phi] = Q; P \triangleq$$
 advice $a[\phi] = \sigma x \cdot \tau y \cdot \pi b \cdot Q; P$
where $\{a, x, y, b\} \cap \operatorname{fn}(Q) = \emptyset$ and $a \notin \operatorname{fn}(P)$

Define *bound* and *free* names as usual. Write $\stackrel{\alpha}{=}$ for the equivalence generated by consistent renaming of bound names, $[\frac{v}{x}]$ for the capture-free substitution of name v for free name x, and $[\vec{m}/a]$ for the capture-free substitution of the name list \vec{m} for free name a. Denote simultaneous substitution as $[\frac{v}{x}, \frac{w}{y}]$.

Pointcuts. The grammar for pointcuts is as follows.

φ,ψ ::=	Pointcut		
true	false	True, False	
φ∧ψ	φ∨ψ	And, Or	
$\forall x \leq p . \phi$	$\exists x \leq p . \phi$	All, Some	
$p \rightarrow q: \ell$	$ \neg p \rightarrow q: \ell$	Atom, Not Atom	

The satisfaction relation, " \vec{D} ; $\vdash p \rightarrow q: \ell \text{ sat } \phi$ ", states that message $p \rightarrow q: \ell$ satisfies ϕ assuming the role hierarchy given by \vec{D} . Satisfaction is defined in the standard way,

building up from the atoms. We say that pointcuts ϕ and ψ overlap in \vec{D} ; if for some *p*, *q* and ℓ , \vec{D} ; $\vdash p \rightarrow q: \ell$ sat ϕ and \vec{D} ; $\vdash p \rightarrow q: \ell$ sat ψ .

We write " $p \cdot \ell$ " for the pointcut which fires when p or one of its subroles receives message ℓ :

$$p.\ell \triangleq \exists x \leq \text{top}. \exists y \leq p. x \rightarrow y: \ell$$

Dynamic Semantics. The reduction relation, $P \rightarrow P'$, is defined by two rules. The first defines *advice lookup*. The second defines *advice invocation*. Advice lookup replaces the message $p \rightarrow q: \ell$ with $p \rightarrow q: \vec{a}$, where \vec{a} is the advice associated with $p \rightarrow q: \ell$. The order in the sequence of advice is the same as the declaration order. The rule treats the rightmost message in a sequence.

$$\vec{D}; |\text{et} z = p \to q : \vec{m}, \ell; P \to \vec{D}; |\text{et} z = p \to q : \vec{m}, \vec{a}; P$$
where $[\vec{a}] = \begin{bmatrix} a \mid (\text{advice } a[\phi] \cdots) \in \vec{D} \\ \text{and } \vec{D}; \vdash p \to q : \ell \text{ sat } \phi \end{bmatrix}$

Advice invocation replaces the message $p \rightarrow q:a$ with the body of *a*. This requires a few substitutions to work. Suppose the body of *a* is " $\sigma x \cdot \tau y \cdot \pi b \cdot Q$ ", where *Q* is " \vec{B} ; return *v*". Suppose further that we wish to execute "let $z = p \rightarrow q: \vec{m}, a; P$ ". The source of the message is *p*, the target is *q*, the body to execute is \vec{B} returning *v*, and the subsequent messages are \vec{m} . This leads us to execute $\vec{B}[P/x, q/y, \vec{m}/b]$ then P[v/z]. The substitution in *P* accounts for the returned value of *Q*. As a final detail, we must take care of collisions between the bound names of *Q* and *P*. We define the notation "let z = Q; P" to abstract the details of the required renaming.

let
$$z = Q; P \triangleq \vec{B}; P[\frac{v}{z}]$$

where $bn(\vec{B}) \cap fn(P) = \emptyset$ and $Q \stackrel{\alpha}{=} \vec{B};$ return v

With this notation, the rule can be written as follows.

$$\vec{D}; |\text{let} z = p \to q; \vec{m}, a; P \to \vec{D}; |\text{let} z = Q[P/x, q/y, \vec{n}/b]; P$$

where (advice $a[\cdots] = \sigma x \cdot \tau y \cdot \pi b \cdot Q \in \vec{D}$

Note that in the reduction semantics, the ordering of advice is significant only for overlapping pointcuts.

Garbage Collection. In the following sections, we present encodings that leave behind useless declarations as the terms reduce. In order to state correctness of the translations, we must provide a way to remove unused declarations from a term. For example, the following rule allows for collection of unused roles:

$$\vec{D}$$
; role $p < q; P \xrightarrow{gc} \vec{D}; P$ where $p \notin fn(P)$

An adequate set of garbage collection rules is given in the full version of the paper [9].

3 Translation of Other AOP Languages into µABC

The small collection of basic orthogonal primitives of μABC make it a viable candidate to serve a role analogous to that of object-calculi in the study of object-oriented programming, provided that it is expressive enough. We establish the expressive power of μABC by *compositional* translations from the following languages that add aspects added on top of distinct underlying programming paradigms:

- A lambda-calculus with aspects core minAML [31].
- An imperative class-based language (in the spirit of Featherweight Java [17], Middleweight Java [8], and Classic Java [14]) enhanced with aspects [19].

On one hand, the translations support our hypothesis that μABC captures a significant portion of the world of aspects. On the other hand, they establish that aspects, in isolation, are indeed a full-fledged computational engine.

In this extended abstract, we discuss only minAML; the encoding of the class-based language is given in the full version [9]. We start with a discussion of functions and conditionals.

3.1 Functions and Conditionals

The encodings in this section rely heavily on the following notation. In a context expecting a program, define "*x*" as the program which returns *x*, and define " $p \rightarrow q : \vec{m}$ " as the program which returns the result of the message:

$x \triangleq \operatorname{return} x$ $p \to q: \vec{m} \triangleq \operatorname{let} x = p \to q: \vec{m}; \operatorname{return} x$

Given this shorthand, we encode abstraction and application as follows, where f and g are fresh roles and "call" and "arg" are reserved roles that are not used elsewhere. The basic idea is to model an abstraction as a piece of advice that responds to "call" — in response to this method, the advice body invokes the argument by emitting "arg" to initiate evaluation of the argument. An application is encoded in a manner consistent with this protocol: in an application, the argument is bound to advice that triggers on "arg".

$$\lambda x \cdot P \triangleq \operatorname{role} f;$$

advice $[f \cdot \operatorname{call}] = \tau y \cdot \operatorname{let} x = y \rightarrow y : \operatorname{arg}; P;$
return f
 $RQ \triangleq \operatorname{let} f = R;$
role $g < f;$
advice $[g \cdot \operatorname{arg}] = Q;$
 $g \rightarrow g : \operatorname{call}$

Example 1. The encoding of $(\lambda x. P)Q$ is " $\vec{D}; g \to g: call$ ", where \vec{D} ; is as follows:

$$\vec{D}$$
;= role f;
advice $a[f . call] = \tau y . let x = y \rightarrow y : arg; P;$
role $g < f$;
advice $b[g . arg] = Q$;

TEAM LING

This term reduces as:

$$(\lambda x . P) Q = \vec{D}; g \rightarrow g : \text{call}$$

$$\rightarrow \vec{D}; g \rightarrow g : a$$

$$\rightarrow \vec{D}; \text{let} x = g \rightarrow g : \text{arg}; P$$

$$\rightarrow \vec{D}; \text{let} x = g \rightarrow g : b; P$$

$$\rightarrow \vec{D}; \text{let} x = Q; P$$

$$\stackrel{\text{gc}}{\rightarrow} \text{let} x = Q; P$$

From here, Q is reduced to a value v, then computation proceeds to $P[\frac{v}{x}]$.

This is the expected semantics of call-by-value application, except for the presence of the declarations \vec{D} , which we garbage collect.

Example 2. We now give a direct encoding of the conditional. The encoding shows one use of advice ordering. Define "if $p \le q$ then P else Q" as the following program, where r is a fresh role and "if" is a reserved role.

```
if p \le q then P else Q \triangleq role r;
advice [\exists x \le top . x \rightarrow r : if] = Q;
advice [\exists x \le q . x \rightarrow r : if] = P;
p \rightarrow r : if
```

Note that P makes no use of its proceed variable, and so if P fires it effectively blocks Q. We can verify the following.

$$\vec{D}$$
; if $p \le q$ then P else $Q \rightarrow * \stackrel{gc}{\rightarrow} \begin{cases} P \text{ if } \vec{D}; \vdash p \le q \\ Q \text{ otherwise} \end{cases}$

3.2 Encoding Core MinAML in µABC

We sketch an encoding into μABC of the function-based aspect language MinAML defined by Walker, Zdancewic and Ligatti [31]. We treat a subset of core MinAML which retains the essential features of the language. Our goal, in this extended abstract, is not to provide a complete translation, but rather to show that the essential features of [31] are easily coded in μABC . In particular, in [31], advice is considered to be a first-class citizen, where here we treat it as second-class.

Core MinAML extends the lambda calculus with:

- The expression new *p*;*P* creates a new name *r* which acts as a hook.
- The expression $\{p : z \rightarrow Q\} \gg P$ attaches the advice $\lambda z \cdot Q$ to the hook *p*. The new advice is executed after any advice that was previously attached to *p*.
- The expression $\{p : z \rightarrow Q\} \ll P$ is similar, except that the new advice is executed *before* any previously attached advice.
- The expression $p\langle P \rangle$ evaluates P and then runs the advice hooked on p.

The encoding into μABC directly follows these intuitions and is as follows, where p is a fresh role and "hook" is a reserved role. The subtle difference between the encoding

of before and after previous advice is a paradigmatic use of the proceed binder in μABC .

new
$$p; P \triangleq$$
 role p ; advice $[p \cdot hook] = \lambda x \cdot x; P$
 $\{p \cdot x \to Q\} \ll P \triangleq$ advice $[p \cdot hook] = \tau z \cdot \pi b \cdot (\lambda x \cdot \text{let } y = Q; (z \to z : b)(y)); P$
 $\{p \cdot x \to Q\} \gg P \triangleq$ advice $[p \cdot hook] = \tau z \cdot \pi b \cdot (\lambda y \cdot \text{let } x = (z \to z : b)(y); Q); P$
 $p\langle P \rangle \triangleq (p \to p : \text{hook}) P$

Example 3. Walker, Zdancewic and Ligatti present the following example. We show the reductions under our encoding. For the purpose of this example, we extend μABC with integers and expressions in the obvious way.

new
$$p$$
; { $p \cdot x_1 \rightarrow x_1 + 1$ } \ll { $p \cdot x_2 \rightarrow x_2 * 2$ } $\gg p\langle 3 \rangle$

This translates to " \vec{D} ; $(p \rightarrow p: hook)$ 3", where \vec{D} ; is:

```
role p;
advice a[p \cdot hook] = \lambda x_0 \cdot x_0;
advice b[p \cdot hook] = \tau z \cdot \pi b \cdot \lambda x_1 \cdot let y_1 = x_1 + 1; (z \rightarrow z : b)(y_1);
advice c[p \cdot hook] = \tau z \cdot \pi b \cdot \lambda y_2 \cdot let x_2 = (z \rightarrow z : b)(y_2); x_2 * 2;
```

and reduction proceeds as follows.

$$\vec{D}; (p \to p: hook) 3$$

 $\rightarrow \vec{D}; (p \to p: a, b, c) 3$
 $\rightarrow \vec{D}; (\lambda y_2. let x_2 = (p \to p: a, b)(y_2); x_2 * 2) 3$
 $\rightarrow^* \stackrel{gc}{\rightarrow} \vec{D}; let x_2 = (p \to p: a, b)(3); x_2 * 2$
 $\rightarrow \vec{D}; let x_2 = (\lambda x_1. let y_1 = x_1 + 1; (p \to p: a)(y_1))(3); x_2 * 2$
 $\rightarrow^* \stackrel{gc}{\rightarrow} \vec{D}; let x_2 = (let y_1 = 3 + 1; (p \to p: a)(y_1)); x_2 * 2$
 $\rightarrow^* \stackrel{gc}{\rightarrow} \vec{D}; let x_2 = (p \to p: a)(4); x_2 * 2$
 $\rightarrow \vec{D}; let x_2 = (\lambda x_0. x_0)(4); x_2 * 2$
 $\rightarrow^* \stackrel{gc}{\rightarrow} \vec{D}; let x_2 = 4; x_2 * 2$
 $\rightarrow \vec{D}; 8$
 $\stackrel{gc}{\rightarrow} 8$

4 Polyadic π -Calculus with Pointcuts

We identify the features required to support an aspect-oriented style by presenting a translation of μABC into a variant of the polyadic π -calculus [26]. The motivation for this portion of the paper is the striking analogies between aspects and concurrency that go beyond our use of concurrency techniques (eg. names, barbed congruences) to study aspects.

Firstly, there are the superficial similarities. Both aspects and concurrency assign equal status to callers/senders and callees/receivers. Both cause traditional atomicity notions to break with the concomitant effects on reasoning — aspects do this by refining atomic method calls into potentially multiple method calls, and concurrency does this by the interleaving of code from parallel processes.

Secondly, there are deeper structural similarities. The idea of using parallel composition to modify existing programs without altering them is a well-understood modularity principle in concurrent programming. Such an analysis is an essential component of the design of synchronous programming languages [6]. Construed this way, the classical parallel composition combinator of concurrency theory suffices for the "obliviousness" criterion on aspect-oriented languages [13] — the behavior of a piece of program text must be amenable to being transformed by advice, without altering the program text.

If this informal reasoning is correct, one might expect that the only new feature that an expressive concurrent paradigm needs to encode μABC is a touch of "quantification", which has been identified as the other key ingredient of the aspect style [13].

Our translation into a polyadic π -calculus is an attempt to formalize this reasoning. Consider a variant of the π -calculus with a hierarchy on names, so the new name process now has the form new x < y; P. We also consider a slight generalization of the match combinator present in early versions of the π -calculus [27], which permits matching on the hierarchy structure on names. The form of the match process is $[\vec{x} \text{ sat } \phi] P$ where ϕ is a formula in a pointcut language that is essentially a boolean algebra built from atoms of the form \vec{x} . The generalized match construct can express traditional (mis)matching via [x = y]P = [x sat y]P.

The dynamics of π is unchanged, apart from an extra rule to handle the generalized matching construct that checks the hierarchy of names (written here as \vec{D} ;) for facts relating to the names (here \vec{z}).

$$\vec{D} \vdash [\vec{z} \operatorname{sat} \phi] P \mapsto P$$
 where $\vec{D} \vdash \vec{z} \operatorname{sat} \phi$

We describe a compositional translation from μABC to the polyadic π -calculus with these mild extensions.

4.1 Syntax and Semantics of π with Pointcuts

Syntax. The grammar for pointcuts is as for μ ABC, except for the atoms.

$$\begin{array}{ll} \phi, \psi ::= & \dots \textit{Pointcut} (As \textit{ for } \mu ABC) \\ \vec{x} & Atom \\ \neg \vec{x} & Not Atom \end{array}$$

The grammar of processes is standard, except for a generalized match construct.

P,Q,R ::=	Process
$z\langle \vec{x} \rangle \mid z(\vec{x}) P$	Output, Input
0 P Q	Termination, Parallel
!P	Replication
new x < y;P	New Name
$[\vec{x} \operatorname{sat} \phi] P$	Match

The matching construct allows for both matching and mismatching. We can define "[x = y] p" as "[x sat y] p" and " $[x \neq y] P$ " as " $[x \text{ sat } \neg y] p$ ".

Dynamic Semantics. Let X range over partially ordered finite sets of names, and write $X \vdash x \leq y$ when $x \leq y$ can be derived from X. The semantics of pointcuts $X \vdash \vec{x} \text{ sat } \phi$ is as for μABC except for the atoms:

$$X \vdash z_1, \dots, z_n$$
 sat z_1, \dots, z_n
 $X \vdash z_1, \dots, z_n$ sat $\neg x_1, \dots, x_m$ if $n \neq m$ or $z_i \neq x_i$ for some i

The dynamic semantics $X \vdash P \rightarrowtail P'$ is given by the usual π -calculus rules, the only difference being that the semantics of pointcuts requires the partial order X in the reduction:

$$\frac{X \vdash \vec{z} \operatorname{sat} \phi}{X \vdash [\vec{z} \operatorname{sat} \phi] P \mapsto P}$$

and so the structural rule for new must include the partial order:

$$\frac{X, x < y \vdash P \rightarrowtail P' \quad x \notin X}{X \vdash \text{new } x < y; P \rightarrowtail \text{new } x < y; P'}$$

The remainder of the dynamic semantics is as given in [26].

4.2 Encoding μABC in π

We now show that μABC can be translated (via a spaghetti-coded CPS transform [29]) into our polyadic π -calculus.

In our translation, following the intuitions expressed in the introduction, advice is simply placed in parallel with the advised code. However, we need to account for a couple of features that disallow the straightforward use of parallel composition and cause the superficial complexity of the translation. First, we need to do some programming to make a single message of interest activate potentially several pieces of advice. Second, the order of invocation of the advice is fixed, so we are forced to program up explicitly the order in which the message is passed down the advice chain.

We will, in fact, translate a sublanguage, but one which contains all programs we consider interesting. A program $P = \vec{D}; Q$ is user code whenever, for any call $p \rightarrow q: \vec{m}$ contained in *P*, we have:

- $-\vec{m}$ is a role ℓ ; or
- \vec{m} is an advice name b bound as a proceed variable that is, there is an enclosing advice declaration advice $a[\phi] = \sigma x \cdot \tau y \cdot \pi b \cdot \cdots$.

Unfortunately, user code is not closed under reduction: if *P* is user code, and $P \rightarrow P'$, then *P'* is not necessarily user code. We defined user closed code, which is closed under reduction.

Definition 4. A program $P = \vec{D}$; *Q* is user closed whenever, for any call $p \rightarrow q$: \vec{m} contained in *P*, we have either:

m is a role *l*; or *m* is an advice name b bound as a proceed variable; or *m* is a sequence "*a*" and for some *b*, r, s and *l*:

$$[\vec{a}, \vec{b}] = \begin{bmatrix} c & \text{advice} c[\phi] \cdots \in \vec{D} \\ \vec{D}; \vdash r \to s : \ell \text{ sat } \phi \end{bmatrix}$$

Let ρ be a partial function from names to quadruples of names. We define the translation \vec{D} ; $\vdash [\![P]\!](k,c,\rho) = Q$ in Table 1. Write " \vec{D} ; $\vdash [\![P]\!] = Q$ " as shorthand for " \vec{D} ; $\vdash [\![P]\!]$ (result, error, \emptyset) = Q".

Table 1. Translation from μABC to π

 $\vec{D}; \vdash [D; P](k, c, \rho) = \text{new } a < c; (Q \mid !a(r, s, \ell, x, y, k', c'))$ $[r, s, \ell \text{ sat } \llbracket \varphi \rrbracket] Q' \\ | [r, s, \ell \text{ sat } \neg \llbracket \varphi \rrbracket] c \langle r, s, \ell, x, y, k', c' \rangle$ where $D = \text{advice } a[\phi] = \sigma x \cdot \tau y \cdot \pi b \cdot P'$ and $\vec{D}; D; \vdash \llbracket P \rrbracket (k, a, \rho) = Q$ and $\vec{D}; D; \vdash [P'](k', c', \rho \cup \{b \mapsto (c, r, s, \ell)\}) = Q'$ \vec{D} ; $\vdash \llbracket D; P \rrbracket (k, c, \rho) = \text{new } p < q; Q$ where $D = \text{role } p < q \text{ and } \vec{D}; D; \vdash \llbracket P \rrbracket(k, c, \rho) = Q$ $\vec{D}; \vdash \llbracket \mathsf{let} x = p \to q : \varepsilon; P \rrbracket(k, c, \rho) = \mathsf{new} \ k' < k; (\mathsf{error} \langle p, q, \ell, p, q, k', c \rangle \mid k'(x, c') \ Q)$ $\vec{D}; \vdash \llbracket \det x = p \rightarrow q : \ell; P \rrbracket(k, c, \rho) = \operatorname{new} k' < k; (c \langle p, q, \ell, p, q, k', c \rangle \mid k'(x, c') Q)$ where \vec{D} ; $\vdash \llbracket P \rrbracket (k, c', \rho) = Q$ $\vec{D}; \vdash \llbracket \text{let} x = p \rightarrow q : b; P \rrbracket(k, c, \rho) = \text{new } k' < k; (d\langle r, s, \ell, p, q, k', c \rangle \mid k'(x, c') Q)$ where $\rho(b) = (d, r, s, \ell)$ and \vec{D} ; $\vdash \llbracket P \rrbracket (k, c', \rho) = Q$. $\vec{D}; \vdash \llbracket \mathsf{let} x = p \rightarrow q; \vec{a}, b; P \rrbracket (k, c, \rho) = \mathsf{new} \ k' < k; (b \langle r, s, \ell, p, q, k', c \rangle \mid k'(x, c') Q)$ where $[\vec{a}, b] = \begin{bmatrix} a \mid dvice a[\phi] \cdots \in \vec{D} \\ \vec{D}; \vdash a \le b \\ \vec{D}; \vdash r \rightarrow s: \ell \text{ sat } \phi \end{bmatrix}$ and \vec{D} ; $\vdash \llbracket P \rrbracket (k, c', \rho) =$ \vec{D} ; $\vdash [[return v]](k, c, \rho) = k \langle v, c \rangle$

The translation uses communication of seven-tuples $\langle r, s, \ell, x, y, k, c \rangle$. Here *r* is the original caller, *s* is the original callee, ℓ is the original method name, *x* is the current caller of a piece of advice, *y* is the current callee, *k* is a continuation *c* is the name of the most recently declared advice. Whenever a method is called, the translation goes through the list *c*, checking advice in order. This encodes advice lookup.

Note that the translation is partial: there exist programs P such that there is no Q for which [P] = Q. However, on user closed programs, the translation is total: there always exists such a Q. Moreover, on user code, the translation is a function: Q is uniquely determined by P.

Theorem 5. For any user code P, if $\llbracket P \rrbracket = Q$ then $P \rightarrow^* \vec{D}$; return v iff $\vdash Q \rightarrow^*$ new $\vec{x} < \vec{y}$; $(Q' \mid \text{result}\langle v \rangle)$.

5 Conclusions and Future Work

 μ ABC was deliberately designed to be a small calculus that embodies the essential features of aspects. However, this criterion makes μ ABC an inconvenient candidate to serve in the role of a meta-language that is the target of translations from "full-scale" aspect languages. There is recent work on such meta-languages (eg. [10] builds on top of the full object calculus), and the bridging of the gap between μ ABC and such work remains open for future study. In this vein, we are exploring the addition of temporal connectives to the pointcut logic of μ ABC. Such an approach provides a principled way to understand and generalize features in existing aspect languages, e.g. cflow in AspectJ, that quantify over sequences of events.

There is ample evidence that aspect-oriented programming is emerging as a powerful tool for system design and development. From the viewpoint of CONCUR, aspects provide two intriguing opportunities. First, the techniques and approaches that have been explored in concurrency theory provide the basis for a systematic foundational analysis of aspects. Our description of μ ABC and its expressiveness falls into this category. In a more speculative vein, the large suite of tools and techniques studied in concurrency theory are potentially relevant to manage the complexity of reasoning required by aspect-oriented programming. Our translation of μ ABC into the pi-calculus is a step in understanding this connection.

References

- 1. AspectJ website. http://www.eclipse.org/aspectj/.
- 2. Martin Abadi and Luca Cardelli. A Theory of Objects. Springer Verlag, 1996.
- M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting objectinteractions using composition-filters. In *In object-based distributed processing, LNCS*, 1993.
- 4. Association of Computing Machinery. Communications of the ACM, Oct 2001.
- Lujo Bauer, Jarred Ligatti, and David Walker. A calculus for composing security policies. Technical Report TR-655-02, Dept. of Computer Science, Princeton University, 2002.
- 6. A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9): 1270–1282, September 1991.
- L. Bergmans. "Composing Concurrent Objects Applying Composition Filtersfor the Development and Reuse of Concurrent Object-Oriented Programs". Ph.d. thesis, University of Twente, 1994. http://wwwhome.cs.utwente.nl/~bergmans/phd.htm.
- G.M. Bierman, M.J. Parkinson, and A.M. Pitts. An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, April 2003.
- G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely. μABC: A minimal aspect calculus. Full version, available at http://fpl.cs.depaul.edu/ajeffrey/papers/muABCfull.pdf, 2004.

- Curtis Clifton, Gary T. Leavens, and Mitchell Wand. Parameterized aspect calculus: A core calculus for the direct study of aspect-oriented languages. Submitted for publication, at ftp://ftp.ccs.neu.edu/pub/people/wand/papers/clw-03.pdf, oct 2003.
- Daniel S. Dantas and David Walker. Aspects, information hiding and modularity. Submitted for publication, at http://www.cs.princeton.edu/~dpw/papers/aspectml-nov03. pdf, 2003.
- 12. Bruno Dufour, Christopher Goard, Laurie Hendren, Clark Verbrugge, Oege de Moor, and Ganesh Sittampalam. Measuring the dynamic behaviour of AspectJ programs, 2003.
- R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness, 2000.
- 14. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In ACM Symposium on Principles of Programming Languages (POPL), pages 171–183,1998.
- Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In 7th International Conference on Concurrency Theory (CON-CUR'96), pages 406–421, Pisa, Italy, 1996. Springer-Verlag. LNCS 1119.
- C. A. R. Hoare. *Communicating Sequential Processes*. Int. Series in Computer Science. Prentice Hall, 1985.
- Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems, 23(3):396–450, May 2001.
- R. Jagadeesan, A. Jeffrey, and J. Riely. A typed calculus for aspect-oriented programs. Submitted for publication, at http://fpl.cs.depaul.edu/ajeffrey/papers/typedABL. pdf, 2003.
- Radha Jagadeesan, Alan Jeffrey, and James Riely. An untyped calculus of aspect oriented programs. In *Conference Record of ECOOP 03: The European Conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, 2003.
- Gregor Kiczales and Yvonne Coady. http://www.cs.ubc.ca/labs/spl/projects/ aspectc.html.
- Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
- Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, 1997.
- 23. K. J. Lieberherr. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, 1996.
- C. V. Lopes. "D: A Language Framework for Distributed Programming". Ph.d. thesis, Northestern University, 1997. ftp://ftp.ccs.neu.edu/pub/people/lieber/theses/ lopes/dissertation.pdf.
- Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in objectoriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
- R. Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.
- Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–40,1992.
- 28. H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, 2001.

- Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. Theoretical Computer Science, 1:125–159,1975.
- David Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In Conference Record of AOSD 03: The 2nd International Conference on Aspect Oriented Software Development, 2003.
- 31. David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Conference Record* of *ICFP 03: The ACM SIGPLAN International Conference on Functional Programming*, 2003.
- 32. Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 2003. To appear.

Type Based Discretionary Access Control*

Michele Bugliesi, Dario Colazzo, and Silvia Crafa

Università Ca' Foscari, Venezia

Abstract. Discretionary Access Control (DAC) systems provide powerful mechanisms for resource management based on the selective distribution of capabilities to selected classes of principals. We study a type-based theory of DAC models for concurrent and distributed systems represented as terms of Cardelli, Ghelli and Gordon's pi calculus with groups [2]. In our theory, groups play the rôle of principals, and the structure of types allows fine-grained mechanisms to be specified to govern the transmission of names, to bound the (iterated) re-transmission of capabilities, to predicate their use on the inability to pass them to third parties, ... and more. The type system relies on subtyping to help achieve a selective distribution of capabilities, based on the groups in control of the communication channels. Type preservation provides the basis for a safety theorem stating that in well-typed processes all names flow according to the delivery policies specified by their types, and are received at the intended sites with the intended capabilities.

1 Introduction

Type systems have been applied widely in process calculi to provide static guarantees for a variety of safety and security properties, including policies for access control [4, 3], non-interference [9, 14, 8], secrecy [1, 2]. In this paper we focus on access control in distributed systems based on Discretionary Access Control (DAC) models [10, 15], i.e. resource management systems that support fine-grained policies for the distribution of capabilities to selected classes of users. To motivate, we start with a well-known example from Pierce and Sangiorgi's seminal paper on types for the pi calculus [13]:

$$S = !s(x).\overline{print}\langle x \rangle$$
 $C = \overline{s}\langle j_1 \rangle.\overline{s}\langle j_2 \rangle...$

S is a print spooler serving print requests from a channel *s*; *C* is a client sending jobs $j_1, j_2, ...$ to the spooler. Given this specification, one may wish to show that each of the jobs sent by *C* is eventually received and printed. While such guarantees can been made for the system S | C, they may hardly be enforced in more general situations. A misbehaved client *C'* may participate to the protocol, as in S | C | C', to steal *C*'s jobs: C' = s(x).s(y).... The capability-based type system developed in [13] prevents this unwanted behavior by requiring that all clients be only granted write capabilities on the channel *s*, and by reserving read capabilities on *s* to the spooler:

$$(\mathsf{v}d:((T)^{\mathsf{rw}})^{\mathsf{rw}})(\mathsf{v}s:(T)^{\mathsf{rw}})(\overline{d}\langle s\rangle \mid \overline{d}\langle s\rangle \mid d(x:(T)^{\mathsf{r}}).S \mid d(y:(T)^{\mathsf{w}}).C)$$

^{*} Work partially supported by EU-FET project 'MyThS' IST-2001-32617.

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 225-239, 2004.

[©] Springer-Verlag Berlin Heidelberg 2004

The types $(T)^{\nu}$, with $\nu \in \{r, w, rw\}$, indicate the types of channels carrying values of type *T* with the associated capabilities for reading, writing, or both. By delivering the channel *s* at different types we can thus enforce an access control policy stating that only the spooler can read jobs.

Notice, however, that the ability of that type system to control the behavior of the system is still rather limited. Indeed, if we want to prevent client jobs from being read by any process other than the spooler *S*, we need to disallow situations like the following:

$$!s(x).\overline{log}\langle x\rangle.\overline{print}\langle x\rangle | \overline{s}\langle j_1\rangle.\overline{s}\langle j_2\rangle | log(y).SPY$$

where the spooler forwards each of the jobs it receives to process *SPY*. The capabilitybased access control from [13] is of little help here, unless one resorts to a more complex encoding of the system or imposes overly restrictive conditions (e.g. prevent the server from writing on all public channels).

A similar problem arises in the following variation of the protocol, in which clients request an ack message to be notified that their jobs have been printed.

$$S = (v print) (!s(x).\overline{print}\langle x \rangle | !print(x).(P | \overline{ack}\langle x \rangle)) \quad C = \overline{s}\langle j_1 \rangle .ack(x).\overline{s}\langle j_2 \rangle .ack(y)$$

As in the previous case, the capability-based type system will fail to detect violations of the intended protocol due to malicious (or erroneous) servers that discard jobs by, say, running the process $s(x).\overline{ack}\langle x \rangle$.

To counter these problems, we propose a novel typing discipline in which we complement the capability-based control system of [13] with a richer class of types that convey information needed to describe, and prescribe, the ways that values may be exchanged within the different system components. The new types have the form $G[T \parallel \Delta]$, where G identifies the authority in control of the values of that type, T describes the structure of those values, and Δ is a *delivery policy* governing the circulation of such values along the channels of the system. To illustrate, the typing

$$j: \mathsf{Job}[\mathsf{file_desc} || \mathsf{Spool} \rightarrow \mathsf{Print} \rightarrow \mathsf{Client}]$$

construes j as a file descriptor to be first delivered to the spooler, then passed on to the printer, and only then re-transmitted back to clients for notification. Equivalently, the delivery policy associated with the type of jobs states that (i) no notification should be given to clients for jobs that have not previously been sent to a printer, and that (ii) no job with such type should be received by the printer unless it has first been delivered to the spooler. Similarly, the typing

$$s: \text{Spool}[(\text{Job}[file_desc || Print \rightarrow \text{Client}])^{rw} || -]$$

defines *s* as a (full-fledged) channel, in control of the spooler, and carrying file descriptors which may be passed on to a client only after having been transmitted to the printer (in addition, the type states that *s* itself should not be re-transmitted). Given the two type assumptions for *j* and *s*, our type system will guarantee that transmitting *j* over *s* is a well-defined, and legal, operation. Remarkably, this requires a non-standard typing of the output prefix, one that guarantees that *j* is received by *s* at the

type $J = Job[file_desc \parallel Print \rightarrow Client]$, so that *j* may only be further re-directed to a printer, as expected.

The type system allows for a wide range of delivery policies to be specified, from policies that support delivery chains of unbounded depth, by resorting to recursively defined types, to policies based on multiple, possibly branching, delivery chains along alternative paths, as in $G[T \parallel G_1 \rightarrow \cdots \rightarrow G_n; G'_1 \rightarrow (G'_2 \rightarrow G'_3; G'_4 \rightarrow (G'_5; G'_6))]$. To illustrate, in our printer example, the initial typing for the channel *s* should be defined as follows (with J as given above):

 $s: \text{Spool}[(J)^{\mathsf{rw}} \parallel \text{Spooler}@(J)^{\mathsf{r}}; \text{Client}@(J)^{\mathsf{w}}].$

This typing guarantees that *s* is received at the expected types, namely with read and write capabilities at the spooler and client sites respectively. Relying on similar typing disciplines, one may guarantee that client jobs remain confined within the printer authority, and thus ensure they are not logged and/or leaked to any spy process.

Furthermore, the capability-based access control from [13], based on subtyping, is still available in our system to selectively advertise values at different types depending on the different principals at which they are delivered, as in

$$\mathsf{G}[T \parallel \mathsf{G}_1 @ T_1 \to \cdots \to \mathsf{G}_n @ T_n ; \mathsf{G}_1' @ T_1' \to (\mathsf{G}_2' @ T_2' \to \mathsf{G}_3' @ T_3' ; \mathsf{G}_4' @ T_4')].$$

Remarkably, the types at the intermediate delivery steps may be different, as long as they are all super-types of the type decided at the originating site.

In the rest of the paper we formalize the approach we have outlined in a typed extension of the pi calculus with groups of [2]. We inherit the syntax of the calculus from [2], and introduce a novel operational semantics to express the flow of names during the computation. We also extend the structure of types to capture the access control policies of interest, and we devise a novel typing system for the analysis and the static detection of violations of such policies. The resulting typing discipline is rather flexible and expressive, as we show by providing several examples of powerful discretionary access control policies formalized in our system. A type preservation theorem, proved for the system, allows us to derive a strong safety result stating that all well-typed processes comply with the discretionary access control policies governing the use of resources.

Plan of the Paper. §2 reviews the pi calculus with groups from [2], introduces the 'flow' semantics and our new classes of types and illustrates their use with a number of examples. §3 describes the typing system. §4 reports on the properties of the type system, on the relationships with the system in [2]. §5 concludes with final remarks and a discussion of related work.

2 The Typed Calculus

The syntax of processes is the same as in the pi calculus with groups [2] (πG for short) summarized below. We presuppose countable sets of names *n*, *m*, ..., and variables *x*, *y*, *z*, reserving *a*, *b*, *c* to range over both sets. We also presuppose a countable set of group

names G_1, G_2, \ldots containing the distinguished group Default. The notions of free names for a process *P*, noted *fn*(*P*), and free groups in a type *T*, noted *fg*(*T*) are just as in πG .

$$P,Q ::= \mathbf{0} \mid c(\tilde{x}:\tilde{\tau}).P \mid \bar{c}\langle \tilde{a} \rangle.P \mid (\mathbf{v}n:\tau)P \mid (\mathbf{v}G)P \mid P|Q \mid P|$$

The novelty with respect to πG is in the structure of types. As in that case, our types are built around group names, but they have a richer structure. Specifically, we interpret groups as representing the authorities (principals) in control of the resources. In addition, drawing inspiration from [12], we structure types so as to convey information on how such resources should be propagated to other principals. The syntax of types is given by the following productions:

Structural Types	$T ::= B \mid (T_1, \ldots, T_n)^{v}$	$(v \in \{r, w, rw\}, \tau_i \ closed)$
Resource Types	$\tau ::= X \mid \mu X.G[T \parallel \Delta \{X\}] \mid G[T \parallel \Delta \{X\}]$]
Delivery Policies	$\Delta ::= [G_i \to T_i]_{i \in I}$	$(G_i = G_j \Rightarrow i = j)$

A structural type conveys structural information on the values with that type, i.e. whether they are basic values, of type B, or communication channels: as in other systems, a channel type specifies the types τ_i of the values transmitted over the channel together with the capabilities v associated with the channel¹.

A resource type is built around a structural type and it additionally specifies the group, or authority, that is in control of the values with that type, together with a set of delivery constraints. A delivery policy, in turn, specifies which other authorities, if any, may legally be granted access to the resource and the extent of the associated access rights (i.e. the capabilities delivered to such authorities). In addition, a delivery policy may impose bounds on the iterated re-distribution of capabilities, and or predicate the delivery of values to the inability to pass them to third parties.

Resource types are recursive, to make it possible to express policies that allow the re-transmission of value to an unbounded depth/distance. Instead, for conceptual simplicity (only), we disallow direct recursion on structural types. We assume type equality up to (*i*) renaming of bound type variables, (*ii*) permutation of delivery constraints inside delivery policies (e.g. $G[T \parallel [G_1 \rightarrow \tau_1; G_2 \rightarrow \tau_2] = G[T \parallel [G_2 \rightarrow \tau_2; G_1 \rightarrow \tau_1])$, and (*iii*) unfolding of recursive types (i.e. $\mu X.G[T \parallel \Delta \{X\}] = G[T \parallel \Delta \{\mu X.G[T \parallel \Delta \{X\}]\}]$). We introduce a number of conventions to ease the notation. We write G[T] for the type $G[T \parallel []$, whose delivery policy is empty, and introduce the following simplified syntax for (finite-depth) delivery chains:

$$G[T \parallel G_1 @ T_1 \to G_2 @ T_2 \to \dots \to G_n @ T_n]$$

$$\triangleq G[T \parallel [G_1 \to G[T_1 \parallel [G_2 \to G[T_2 \parallel [\dots [G_n \to G[T_n]] \dots]]]]]$$

$$G[T \parallel G_1 \to G_2 \to \dots \to G_n] \triangleq G[T \parallel G_1 @ T \to G_2 @ T \to \dots \to G_n @ T]$$

¹ In principle, capabilities may be defined to control the access and use of values of any type. We restrict to channel capabilities for presentation purposes only.

2.1 Operational Semantics

The intention of the type system is to control the flow of names, so as to provide static guarantees against any leakage of names to unintended users and/or at unintended types. Expressing flows is subtle, however, because different occurrences of the same name may flow along different paths during the computation. To illustrate, consider a type $\tau = G[B \parallel G_1 \rightarrow G_2; G_3]$ describing values of basic type *B* to be delivered either to G_1 and then on to G_2 , or to G_3 (always at the structural type B). Assume, further, that we are given the following two processes where $n_1:G_1[...], n_2:G_2[...], n_3:G_3[...]$ and $m:\tau$:

$$P \triangleq \overline{n_1} \langle m \rangle \mid \overline{n_3} \langle m \rangle \mid n_1(x) . n_3(y) . \overline{n_2} \langle x \rangle, \quad Q \triangleq \overline{n_1} \langle m \rangle \mid \overline{n_3} \langle m \rangle \mid n_1(x) . n_3(y) . \overline{n_2} \langle y \rangle$$

Then, *P* should be judged safe, as both copies of *m* flow along legal paths, while *Q* should be deemed unsafe, as it allows an illegal flow for *m*. However, with the standard reduction semantics these judgments may hardly be made, as after two reduction steps both *P* and *Q* reduce to $\overline{n_2}(m)$.

To make the notion of flow explicit in the calculus, we resort to a non-standard semantics that uses tags for each name to trace the sequence of channels traversed by the name during the computation. To illustrate, the tagged name $n_{[mpq]}$ represents the name *n* that flowed first through the channel *m*, then through *p* and finally through *q*. Here '*mpq*' is short for the extended notation $m :: p :: q :: \varepsilon$, where ε denotes the empty sequence and '::' is the usual right-associative sequence constructor. We let φ range over sequences of names, and use the notation $\varphi \cdot n$ to indicate the sequence resulting from appending the name *n* to the tail of φ .

The resulting reduction relation is defined over the class of *dynamic* processes, noted *A*, *B*, The structure of dynamic processes coincides with the structure of processes, except that the former may use tagged names in addition to names. Indeed, processes may be understood as special cases of dynamic processes in which all names are tagged with the empty name sequence (e.g. $n_{[\epsilon]}$, that we identify with *n*). The notion of free names for dynamic processes is redefined to account for presence of the tags. Specifically, $fn(n_{[\phi]}(\tilde{x} : \tilde{\tau}) \cdot A) = fn(A) \cup \{n\} \cup \{m \mid m \in \phi\}$, and similarly for the remaining (dynamic) process constructs.

The dynamics of the calculus, in Table 1, is defined as usual in terms of an auxiliary relation of structural congruence. Structural congruence is exactly as in the πG , but relies on the different definition of free names discussed above. The core of the reduction relation is in the (*red comm*) rule, which updates the flow tags to reflect the flow of each of the arguments through the synchronization channel. With this notion of reduction we may now judge process *P* above safe, and *Q* unsafe, as the two reduction sequences

$$P \longrightarrow \overline{n_2} \langle m_{[n_1]} \rangle$$
 and $Q \longrightarrow \overline{n_2} \langle m_{[n_3]} \rangle$

exhibit the different flow of *m* in the two computations. We remark here that the tags are only instrumental to record the flow of each name, and have no further effect on the reductions available for processes. We make this precise by relating our flow-sensitive semantics of Table 1 with the original reduction semantics of πG . The latter, which we denote with \mapsto , is defined on processes (rather than on dynamic processes) exactly as we do here, but uses the standard communication rule, namely:

$$\overline{n}\langle m_1,\ldots,m_k\rangle P \mid n(x_1:\tau_1,\ldots,x_k:\tau_k) Q \mapsto P \mid Q\{m_i/x_i\}$$

TEAM LING

Dynamic Processes: a, b, \ldots denote either variables of tagged names of the form $n_{[\omega]}$.

$$A ::= \mathbf{0} \mid a(x_1: \mathbf{T}_1, \dots, x_n: \mathbf{T}_n) \cdot A \mid \overline{a}(b_1, \dots, b_n) \cdot A \mid (\mathbf{v}n: \mathbf{T})A \mid (\mathbf{v}G)A \mid A \mid B \mid B$$

Structural congruence

(monoid))	$A \mid B \equiv B \mid A, A \mid 0 \equiv A, (A \mid B) \mid 0$	$C \equiv A \mid (B \mid C)$
(repl))	$A \equiv A A$	
(name extr)	$(\mathbf{v}n:\mathbf{T})(A\mid B)\equiv A\mid (\mathbf{v}n:\mathbf{T})B$	if $n \notin fn(A)$
(group extr)	$(vG)(A \mid B) \equiv A \mid (vG)B$	if $G \notin fg(A)$
(name/group exch)	$(\mathbf{vG})(\mathbf{v}n:\mathbf{T})A \equiv (\mathbf{v}n:\mathbf{T})(\mathbf{vG})A$	if $G \notin fg(\tau)$
(name exch)	$(\mathbf{v}n_1:\mathbf{T}_1)(\mathbf{v}n_2:\mathbf{T}_2)A \equiv (\mathbf{v}n_2:\mathbf{T}_2)(\mathbf{v}n_1:\mathbf{T}_1)A$	
(group exch)	$(\mathbf{v}\mathbf{G}_1)(\mathbf{v}\mathbf{G}_2)A \equiv (\mathbf{v}\mathbf{G}_2)(\mathbf{v}\mathbf{G}_1)A$	

Reduction

(red comm)	$\overline{n_{[\varphi]}}\langle m_{1[\varphi_{1}]},\ldots,m_{k[\varphi_{k}]}\rangle A \mid n_{[\psi]}(x_{1}:\mathcal{T}_{1},\ldots,x_{k}:\mathcal{T}_{k}).B \longrightarrow A \mid B\{ \stackrel{m_{i[\varphi_{i},n]}}{\longrightarrow}/_{x_{i}} \}$
(red res)	$A \longrightarrow A' \implies (\mathbf{v}n:\mathbf{T})A \longrightarrow (\mathbf{v}n:\mathbf{T})A'$
(red group)	$A \longrightarrow A' \implies (vG)A \longrightarrow (vG)A'$
(red par)	$A \longrightarrow A' \Longrightarrow A \mid B \longrightarrow A' \mid B$
(red struct)	$A \equiv \longrightarrow \equiv B \Longrightarrow A \longrightarrow B$

in place of our (*red comm*) rule. Given any dynamic process A, let |A| denote the (proper) process resulting from erasing all the tags from A. Then we have:

Theorem 1 (Flow Reduction vs Standard Reduction). Let A be a closed dynamic process. If $A \longrightarrow^* B$ then $|A| \mapsto^* |B|$. Conversely, if $|A| \mapsto^* Q$ then there exists B such that $A \longrightarrow^* B$ and $|B| \equiv Q$.

This result would hold also in the presence of a matching operator to compare names. In particular, similarly to our (*red comm*) rule, in dynamic processes the matching construct would disregard the tags to decide name equality, as in the following reductions:

$$\begin{array}{l} \text{if } [n_{[\varphi_1]} = n_{[\varphi_2]}] \text{ then } P \text{ else } Q \longrightarrow P \\ \\ \text{if } [n_{[\varphi_1]} = m_{[\varphi_2]}] \text{ then } P \text{ else } Q \longrightarrow Q \quad (n \neq m) \end{array}$$

Besides being coherent with our current development, disregarding the tags to test name equality is crucial to encode any sensible form of matching. The simplest illustration is probably the case of nonce-based authentication protocols, in which a principal generates a nonce (i.e. a fresh name) and then uses matching to test the freshness of an incoming message by comparing the nonce included in the message with the name it generated: clearly, this test may only succeed if we disregard the nonce's flow.

2.2 Types and Discretionary Access Control Policies

To start illustrate of our types, let pwd be a basic type describing passwords. Then we may define the type $\mu X.G[pwd \parallel G \rightarrow X]$ to constrain the free re-transmission of values with this type only within members of group G. Alternatively, we may define the type $\mu X.G[pwd \parallel G \rightarrow X ; F \rightarrow G[pwd]]$ to qualify passwords that may also be passed to friends, of group F, provided that they do not pass them over to third parties.

The re-transmission of values may also be filtered on the basis of the capabilities that are passed along with the values. For instance, given $\tau = \mu Y.G[(nat)^w \parallel F \rightarrow Y]$, we may define the type $\mu X.G[(nat)^{rw} \parallel G \rightarrow X; F \rightarrow \tau]$ to qualify nat channels of group G that may be received and re-transmitted at group F at a type that restricts their use as to write-only channels². Similarly, the type

$$\mu X.\mathsf{G}[(\mathsf{nat})^{\mathsf{rw}} \parallel [\mathsf{G}_1 \to \mathsf{G}[(\mathsf{nat})^{\mathsf{w}} \parallel \Delta_1] ; \quad \mathsf{G}_2 \to \mathsf{G}[(\mathsf{nat})^{\mathsf{r}} \parallel \Delta_2] ; \quad \mathsf{Default} \to X]]$$

qualifies nat channels owned by G, that can be delivered to group G_1 and G_2 as writeonly and read-only channels respectively. Instead, no restriction applies for other groups, as indicated by the Default entry in the delivery policy. The type also indicates the delivery policies Δ_1 and Δ_2 , that we leave unspecified here, for the subsequent 'hops', from G_1 and G_2 respectively. We further discuss the import of resource types in formalizing examples of DAC policies below.

Most forms of discretionary access control focus on the so called *owner-based* administration of access rights, by which the owner of an object, typically its originator, has the discretionary authority over who else can access that object. DAC policies vary depending on how the owner's discretionary power can be delegated to other users.

In *strict DAC* ([15,16]), the owner is the only entity to have the authority to grant access to an object. Such policies are directly expressed in our type system with types with a rather regular structure, namely $\text{Owner}[T \parallel [\text{User}_i \rightarrow \tau_i]_{i \in I}]$ where τ_i is the resource type that constrains the re-transmission of its values only within the authority of User_i, namely: $\tau_i = \mu X.\text{Owner}[T \parallel \text{User}_i \rightarrow X].$

Liberal DAC ([15,16]), models are more flexible, and interesting. They are based on a decentralized authorization policy where the owner of an object delegates other users the privilege of specifying authorizations, possibly with the ability of further delegating it. Two popular classes of Liberal DAC policies are those known as *originator controlled* [11], and *true delegation* [15].

An *originator controlled* policy prevents access to data being extended to any authority without the owner's explicit permission. In this case when a resource is created the owner has full control over how the resource's capabilities can be distributed. An example of such policies is given the diagram below, representing an Owner that creates a channel and specifies how it should be distributed to two parties, Alice and Carl:



² Here, and elsewhere, we say *transmit at a group* to mean *transmit at channels of that group*.

The channel is first delivered, in read mode, to Alice, who is delegated to re-transmit it to Carl with the additional write capability; only then is Alice allowed to receive the write capability from Carl. This delivery policy, imposed by the owner to ensure that Alice will not write on the new channel until it has been received also by Carl, may be expressed with our types as follows:

$$\mathsf{Owner}[(T)^{\mathsf{rw}} \parallel \mathsf{Alice}^{@}(T)^{\mathsf{r}} \rightarrow \mathsf{Carl}^{@}(T)^{\mathsf{rw}} \rightarrow \mathsf{Alice}^{@}(T)^{\mathsf{w}}]$$

A similar example can be recovered from the literature on cryptographic protocols. Consider the case where two parties, Alice and Bob, wish to establish a private exchange. To accomplish that, Alice creates a fresh name, say c_{AB} , sends it to a trusted Server and delegates it to forward the name to Bob so that the exchange may take place. Here, the Server should only act as a forwarder, and not interfere with the exchanges between Alice and Bob. This can be achieved using the typing

$$c_{AB}$$
: Alice[(data)^{rw} || Server@(data) \rightarrow Bob@(data)^{rw}]

in which the the Server receives the channel with no capability, as intended.

True delegation policies are more liberal, and allow any principal receiving the grant option to further distribute it to principals not explicitly anticipated by the owner. Such policies are formalized in our system with the help of **Default** entries. To illustrate, a policy in which the owner principal gives full discretionary power to a delegate, may be expressed by the type **Owner**[$T \parallel D \rightarrow \tau$], where $\tau = \mu X$.**Owner**[$T \parallel$ **Default** $\rightarrow X$].

3 Typing and Subtyping

Having illustrated how types may be employed to formalize discretionary policies, we now turn to the problem of analyzing the import of such policies on the dynamic behavior of processes. Notice, to this regard, that the delivery policies we outlined in the previous section rely critically on the ability to deliver names at types that may vary non-monotonically along the delivery chains. Then, to have the desired safety guarantees, we must envision a mechanism to ensure that the types at each delivery site be a safe approximation of the type at the originating site. This is accomplished by the rules for type formation and subtyping we discuss next.

3.1 Type Formation and Subtyping

Type formation is defined in Table 2. Any type must be built around group names and type variables known to the type environment. In addition any resource type, say $G[T \parallel \Delta]$, must be so defined as to ensure that each resource type occurring in the delivery policy Δ is built around (i) the group name **G** of the owner, and (ii) a structural super-type of the structural type *T*. The first constraint could be lifted, but we do not explore this possibility here; the second constraint, instead, is critical for soundness as we just observed. The type formation rules enforce both constraints with the help of the auxiliary judgments $\Gamma \vdash_{G,T} \tau$, that verify τ in relation to the group **G** and the type *T*, for all the resource types τ introduced by Δ .

Resource Types – owner					
$(\tau\text{-owner})$ $\Gamma \vdash_{G,T} \tau_i fg(G[T \parallel [G_i \to \tau_i]]) \subseteq I$	$Dom(\Gamma)$	$(\tau - var)$ $X \in \Gamma$	$\Gamma \vdash \diamond$	$(\tau\operatorname{-REC})$ $\Gamma, X \vdash G[T \parallel \Delta\{X\}]$	
$\Gamma \vdash G[T \parallel [G_i ightarrow t_i]_{i \in I}]$		$\Gamma \vdash X$		$\Gamma \vdash \mu X.G[T \parallel \Delta \{X\}]$	
Resource Types – step					
$(\tau\text{-STEP})$ $\Gamma \vdash T \leq T' \Gamma \vdash_{G,T} \tau_i$	$(\tau - vAR)$ $X \in \Gamma$) Γ⊢ ◊	$(\tau - \operatorname{REC})$	$G_{,T'} G[T \parallel \Delta\{X\}]$	
$\Gamma \vdash_{G,T} G[T' \parallel [G_i \to T_i]_{i \in I}]$	Γ⊢ _{G,}	TX	$\Gamma \vdash_{G,T'}$	$\mu X.G[T \parallel \Delta \{X\}]$	

Table 2. Type Formation

The subtyping relation is axiomatized in Table 3. The first two blocks of rules are standard (cf. [13]). The core of the subtyping relation is in the last two rules. Rule (**T-TYPE**) makes resource-subtyping covariant in the component structural types, as expected, and required for soundness. More interestingly, (**T-POLICY**) requires resource types to impose more restrictive delivery policies than their sub-types. At a first look, the ordering relation on delivery policies is reminiscent of the subtype relation on record types. This is indeed the case if we restrict to delivery policies without Default entries and predicate $\Delta \preccurlyeq \Delta'$ to the additional constraint that $Dom(\Delta') \subseteq Dom(\Delta)$. The resulting \preccurlyeq relation captures a form of originator controlled DAC model, in which resource owners have full delivery control over their resources. On the other hand, in case $Dom(\Delta') \supset Dom(\Delta)$, and similarly with policies that include Default entries, the ordering relation on policies captures DAC models with *true delegation*, in which intermediate users of a resource may autonomously make decisions on the next delivery steps, provided that they advertise the resource at structural super-type of the owner's structural type.

To illustrate, consider first $\Delta = [Default \rightarrow G[\tau^{rw}]]$ and $\Delta' = [G_1 \rightarrow G[\tau']; G_2 \rightarrow G[\tau^w]]$. For a proper Γ one has $\Gamma \vdash \Delta \preccurlyeq \Delta'$, which provides support for true delegation as Δ allows any delivery, while Δ' distributes the read capability to channels of group G_1 and the write capability to channels of group G_2 . As a further example, consider $\Delta = [G_1 \rightarrow G[\tau']; Default \rightarrow G[\tau^{rw}]]$ and $\Delta' = [Default \rightarrow G[\tau^{rw}]]$. Here Δ allows all groups but G_1 to receive the write capability, whereas Δ' does not make this distinction. The two policies are not related by the \preccurlyeq relation, as $\Gamma \vdash \Delta \preccurlyeq \Delta'$ requires that in the presence of Default entries in both Δ and Δ' , all delivery constraint expressed in Δ must also be enforced by Δ' .

3.2 Typing of Processes and Dynamic Processes

The typing rules for (dynamic) processes, in Table 4, complete the presentation of the type system. We remark that the typing rules validate dynamic processes, hence also

Table	3.	Sub	types
-------	----	-----	-------

(Re	FLEX)	(TRANS)	
Г⊢	$\diamond fg(\mathbf{T}) \subseteq Dom(\Gamma)$	$\Gamma \vdash \tau \leq \rho$	$\Gamma \vdash \rho \leq \sigma$
	$\Gamma \vdash \tau \leq \tau$	Γ⊢τ	$\leq \sigma$
Structural subtypes:			
(T-WRITE)		(T-READ)	
$\Gamma \vdash \tau'_1 \leq \tau_1 \cdots I$	$\Gamma \vdash \mathcal{T}'_n \leq \mathcal{T}_n n \geq 1$	$\Gamma \vdash \tau_1 \leq \tau_1'$	$\cdots \Gamma \vdash \mathbf{T}_n \leq \mathbf{T}'_n n \geq 1$
$\Gamma \vdash (\tau_1, \ldots, \tau_n)^{w}$	$\leq (\tau'_1, \ldots, \tau'_n)^w$	$\Gamma \vdash (\tau_1, .$	$(\ldots, \tau_n)^{r} \leq (\tau'_1, \ldots, \tau'_n)^{r}$
	(T-Read/write) $\Gamma \vdash \diamond$	$n \ge 0$	
	$\Gamma \vdash (\tau_1, \ldots, \tau_n)^{rw}$	$\leq (\tau_1,\ldots,\tau_n)$	v
Resource Subtypes			
Let $\Gamma \vdash \Delta \preccurlyeq \Delta' \triangleq$			
$\forall G \in Dom(\Delta') \text{ if } (G)$ and if $Default \in Dd$	$f \in Dom(\Delta)$) then $\Gamma \vdash \Delta(Dom(\Delta) \cap Dom(\Delta'))$ then $\forall G$	$\mathbf{G}) \leq \Delta'(\mathbf{G}) \text{ els} \\ \mathbf{G} \in Dom(\Delta) \setminus L$	e $\Gamma \vdash \Delta(Default) \leq \Delta'(G)$ $\operatorname{Dom}(\Delta').\Gamma \vdash \Delta(G) \leq \Delta'(Default)$

(T-TYPE)	$(\tau$ -Policy)
$\Gamma \vdash T \leq T' fg(G[T \parallel \Delta]) \cup fg(T') \subseteq Dom(\Gamma)$	$\Gamma \vdash \Delta \preccurlyeq \Delta' fg(G[T \parallel \Delta]) \cup fg(\Delta') \subseteq Dom(\Gamma)$
$\Gamma \vdash G[T \parallel \Delta] \;\; \leq \;\; G[T' \parallel \Delta]$	$\Gamma \vdash G[T \parallel \Delta] \leq G[T \parallel \Delta']$

the (proper) processes of the source calculus. This is required for subject reduction, as the result of a reduction (sequence) is a dynamic process rather than a process.

The typing rules for names allow each name to be typed at (any super-type of) the resource type τ known to the environment, by rules (PROJECT) and (SUBSUMPTION), as well as at any type mentioned at the subsequent hops of each chain of the delivery policy associated with τ . One application of rule (DELIVERY) reaches the types at the first hops; subsequent applications reach types occurring further on along the chains.

This ability to type names at all their delivery types is crucial to the proof of subject reduction. To see why, we first look at rule (OUTPUT), the core of the delivery discipline. Consider the case when name, say *m* is emitted on a channel, say $n : G[(\tau)^w]$. Assume further that *m* is known to the environment at type $F[T \parallel \Delta]$. Rule (OUTPUT) verifies that G is indeed one of the next 'hops' in Δ , and the type τ' at which *m* should be delivered to G is a subtype of τ , the type of values carried by *n*. Given that the types of names known to the environment must be well-formed, the type formation rules ensure that the original structural type of *m* is a subtype of the structural component of τ' , thus

(Емрту)	$\begin{array}{l} (\text{Env} n) \\ \Gamma \vdash \diamond \Gamma \vdash \tau \end{array}$	$ \begin{array}{ll} (E_{NV} x) \\ (E_{NV} x) \\ (F \vdash \tau n \notin Dom(\Gamma) \\ \Gamma \vdash \diamond fg(\tau) \subseteq Dom(\Gamma) x \notin Dom(\Gamma) \\ \end{array} $			
ø⊢◊	$\Gamma, n: \tau$	¦⊢ o	$\Gamma, x: \mathbf{T} \vdash \diamond$		
$\begin{array}{l} (Env G) \\ \Gamma \vdash \diamond G \notin Dor \end{array}$	(ENV) $n(\Gamma) \qquad \Gamma \vdash \diamond$	$X \notin Dom(\Gamma)$	(PROJECT) (SUBSUMPTION) $\Gamma, a: \tau \vdash \diamond$ $\Gamma \vdash a: \tau' \Gamma \vdash \tau' \leq \varepsilon$		
Γ,G⊢◊	I	$T, X \vdash \diamond$	$\Gamma, a: \mathbf{T} \vdash a:$	τ Γι	$-a: \mathbf{T}$
(Delivery) $\Gamma \vdash n_{[\varphi]} : G[T \parallel \Delta]$	$\Gamma \vdash m : G_1[T_1$	$\ \Delta_1$] (G ₁ \rightarrow 1	$C \in \Delta$) \vee (G ₁	$\notin Dom(\Delta) \wedge De$	$fault \to \tau \in \Delta)$
$\begin{split} \Gamma \vdash n_{[\varphi m]} : \mathfrak{T} \\ Let \ \Gamma \vdash \Delta_i(G) \preccurlyeq \mathfrak{T}_i &\triangleq \text{ if } (G \in Dom(\Delta_i)) \text{ then } \Gamma \vdash \Delta_i(G) \leq \mathfrak{T}_i \\ \text{ else } (Default \in Dom(\Delta_i)) \text{ and } \Gamma \vdash \Delta_i(Default) \leq \mathfrak{T}_i \\ (\text{INPUT}) \\ \underline{\Gamma \vdash a : G[(\mathfrak{T}_1, \dots, \mathfrak{T}_k)^r \parallel \Delta] \Gamma, x_1 : \mathfrak{T}_1, \dots, x_k : \mathfrak{T}_k \vdash A 0 \leq k} \end{split}$					
$\Gamma \vdash a(x_1: \tau_1, \ldots, x_k: \tau_k).A$					
$(OUTPUT)$ $\Gamma \vdash a : G[(\tau_1, \dots, \tau_k)^{w} \parallel \Delta] \Gamma \vdash A \Gamma \vdash b_i : G_i[T_i \parallel \Delta_i] \Gamma \vdash \Delta_i(G) \preccurlyeq \tau_i 0 \le i \le k$ $\Gamma \vdash \overline{a} \langle b_1, \dots, b_k \rangle A$					
(New Γ,G⊢	$G \qquad (\text{New } n) \\ f A \qquad \Gamma, n : \tau$	(PAR) $\Gamma \vdash A$	$\frac{\Gamma \vdash B}{\Gamma} = \frac{\Gamma}{\Gamma}$	Dead) (Rep ⊢ ◊ Γ⊢ Λ	L) 4
Γ⊢ (v	G)A $\Gamma \vdash (vn)$	$: \tau)A \qquad \Gamma \vdash A$	$A \mid B \qquad \Gamma$	⊢0 Γ⊢‼	A

Table 4. Typing of dynamic processes. (note: we identify $n_{[\varepsilon]}$ with n)

also a subtype of the type τ at which *n* expects to receive its values: this guarantees that *m* may safely be received at *n*. Further re-transmissions of *m* will undergo the same checks by the (OUTPUT) rule, but now with the types advertised at the subsequent hops in Δ : to prove subject reduction we therefore need to be able to type *m* at all such types. Except for this specificity in the typing of names, the proof of subject reduction for the system is standard (cf. §4).

We give an example to show the effect of value propagation with a typed version of the print spooler discussed in the introduction. Let $!s(x:\tau_1).\overline{print}\langle x \rangle | !print(x : \tau_2).(P | \overline{ack}\langle x \rangle) | (v j:\tau_J) \overline{s}\langle j \rangle.ack(x:\tau_3).C$ be a system where a client creates a new job *j*, sends it to the printer and waits for notification. The desired delivery policy for *j* requires the job to be sent first to the print spooler, then to the printer, and finally back to the client. Such policy is enforced with the types:

$$\begin{split} \tau_{J} &= \mathsf{Job}[\mathsf{file_desc} \parallel \mathsf{Spool} \to \tau_{1}] \quad \tau_{1} = \mathsf{Job}[\mathsf{file_desc} \parallel \mathsf{Print} \to \tau_{2}] \\ \tau_{2} &= \mathsf{Job}[\mathsf{file_desc} \parallel \mathsf{Client} \to \tau_{3}] \quad \tau_{3} = \mathsf{Job}[\mathsf{file_desc} \parallel \Delta] \end{split}$$

and assuming $s : \text{Spool}[(\tau_1)^{\text{rw}}] \text{ print} : \text{Print}[(\tau_2)^{\text{rw}}]$, and $ack : \text{Client}[(\tau_3)^{\text{rw}}]$. We leave it to the reader to verify that all types involved are well-formed and that the process typechecks.

4 Type System Properties

We conclude the presentation of the type system by elucidating its main properties. To state such properties, we first introduce two additional (partial) operators on types and type environments. Given a type environment Γ and a sequence of names φ we let $\Gamma[\varphi]$ denote the sequence of groups that Γ associates with the names in φ .

$$\Gamma[\varepsilon] \triangleq \varepsilon$$

$$\Gamma[n::\phi] \triangleq \text{ if } (\Gamma(n) = \mathsf{G}[T \parallel \Delta] \text{ and } \Gamma[\phi] \neq \bot) \text{ then } \mathsf{G}::\Gamma[\phi] \text{ else } \bot$$

Given a closed resource type τ , and a sequence of group names π , we then denote with $\tau \downarrow \pi$, the type occurring in τ at π , in the following sense:

$$\begin{split} \tau \downarrow \varepsilon &\triangleq \tau \\ G[T \parallel \Delta] \downarrow G' :: \pi &\triangleq \text{ if } (G' \in Dom(\Delta)) \text{ then } \Delta(G') \downarrow \pi \\ &\quad \text{ else if } (\text{Default} \in Dom(\Delta)) \text{ then } \Delta(\text{Default}) \downarrow \pi \text{ else } \bot \end{split}$$

The next three theorems express the safety properties of the type system Rather than defining an explicit notion of error and showing that well-typed terms don't have error transitions, as in, e.g. [4], we state our safety properties directly in terms of types: the two approaches are essentially equivalent. Theorem 2 states that in all well-typed dynamic processes, any access to a channel complies with the access rights associated with the channel, and the arguments are passed over the channel at subtypes of the expected types. Theorem 3 states that in all well-typed dynamic processes names flow according to the delivery policies expressed by their types. Finally, by Theorem 4 we know that such properties are preserved by reduction. Collectively, these theorems provide static guarantees that in well-typed (proper) processes, all resources are accessed and delivered according to the policies defined by their types.

Theorem 2 (Access Control). Let A and B be closed, dynamic processes with $A \equiv \overline{n_{[\phi]}}\langle a_1, \ldots, a_k \rangle A'$ and $B \equiv n_{[\phi']}(x_1:\rho_1, \ldots, x_l:\rho_l) B'$. Assume, further, that $\Gamma \vdash A \mid B$. Then l = k and one has:

$$- \Gamma \vdash n_{[\varphi]} : \mathsf{G}[(\tau_1, \dots, \tau_k)^{\mathsf{w}}], \Gamma \vdash n_{[\varphi']} : \mathsf{G}[(\rho_1, \dots, \rho_k)^r], and - for all i = 1, ..., k, \Gamma \vdash a_i : \sigma_i \text{ with } \sigma_i \downarrow \mathsf{G} \neq \bot, \Gamma \vdash \sigma_i \downarrow \mathsf{G} \leq \tau_i \text{ and } \Gamma \vdash \tau_i \leq \rho_i.$$

Theorem 3 (Flow Control). Let $\Gamma \vdash A$ with A closed. Assume, further, that $\Gamma \vdash A$ depends on the judgment $\Gamma' \vdash n_{[\phi]} : \tau$. Then $n \in Dom(\Gamma')$ and $\Gamma'[\phi] \neq \bot$. In addition, $\Gamma'(n) = \rho$ with p such that $\rho \downarrow \Gamma'[\phi] \neq \bot$ and $\Gamma' \vdash \rho \downarrow \Gamma'[\phi] \leq \tau$.

Theorem 4 (Subject Reduction). *If* $\Gamma \vdash A$ *and* $A \longrightarrow^* B$ *, then* $\Gamma \vdash B$ *.*

We remark that theorem 3 could not be meaningfully stated without appealing to the flow tags attached to names. In particular, it is not true, in general, that given an extended process A, $\Gamma \vdash A$ implies $\Gamma \vdash |A|$. To see that, note that $\Gamma \vdash A$ may depend on two tagged names $n_{[\varphi_1]}$ and $n_{[\varphi_2]}$ being given different types, (not related by subtyping) by resorting to the (DELIVERY) rule. By erasing the tags, we lose the possibility of appealing to the (DELIVERY) rule, and consequently the judgement $\Gamma \vdash |A|$ fails. For this very reason, Theorem 4 does not hold, in general, under the reduction semantics \mapsto of [2]. Interestingly, however, we can recover subject reduction for \mapsto provided that we make adequate assumptions on the structure of the types occurring in Γ and A. We formalize the relationship with the type system of [2] below.

4.1 Encoding the Pi-Calculus with Groups

As we mentioned, the syntax of the pi-calculus with groups is the same as the one in §2. The types, instead, are defined simply as follows: $T ::= G[T_1, ..., T_n]$. These types may be encoded into our resource types as follows:

$$[\mathsf{G}[T_1,\ldots,T_n]] = \mu X \cdot \mathsf{G}[([T_1],\ldots,[T_n])^{\mathsf{rw}} \parallel \mathsf{Default} \to X]$$

The encoding provides all types with the most liberal delivery policy, one that allows unboundedly-deep delivery over all channels: the only constraint is that the receiving channels have access to the group of the value they carry with them, exactly as in πG .

We show that our type system is a conservative extension of the type system of πG . Given a πG type environment Γ and a πG process *P*, let $[\Gamma]$ and [P] be the type environment and process that result from applying the encoding of types systematically to all types occurring in Γ and *P*. Then we have:

Theorem 5 (Relationships with πG). $\Gamma \vdash P$ is derivable in πG iff $[\Gamma] \vdash [P]$ is derivable in our type system.

This follows by observing that if we restrict to simple types, the type formation rules as well as the (OUTPUT) rule for processes coincide with the corresponding rules in πG . Now, call a type *simple* when it is the encoding of a πG type. Similarly, call a type environment and a (dynamic) process *simple* when all the types occurring therein are simple. If Γ and A are simple, it is not difficult to see that $\Gamma \vdash A$ implies $\Gamma \vdash |A|$. Intuitively, the reason is that simple types are insensitive to flows: this is a consequence of the delivery type being the same at all hops in a simple type. More interestingly, for simple processes we have subject reduction, as we state next.

Theorem 6 (Subject Reduction for Simple Processes). Assume $\Gamma \vdash P$ with Γ simple, and P closed and simple, and let $P \mapsto^* Q$. Then $\Gamma \vdash Q$.

Based on this result, the secrecy theorem of [2] can be re-established in our system with no additional effort for simple processes.
5 Conclusion

We have developed a type theory for the specification and the static analysis of access control policies in the pi calculus. Our approach extends and complements previous work on the subject by introducing a new class of types so defined as to control the dynamic flow of values among system components. We have shown the flexibility of our system with several examples, and proved that it provides strong safety guarantees for all well-typed processes.

There are several desirable extensions to the present system: they include the ability to express the revocation of capabilities, to change the ownership on resources, to account for hierarchical relationships among principals, or for declassification mechanisms. A further topic of investigation is the study of typed equivalences to gain deeper insight into the import of our access control policies in the behavioural properties of processes. We leave all these to our plans for future work, and conclude here with a brief discussion on related work.

While we are not aware of any approach specifically targeted at the access control mechanisms we have discussed here, our work is clearly related to a large body of literature on (type-based) security in process calculi.

Several type system encompass various forms of access control policies in distributed systems. Among them, [3] proposes a form of distributed access control based on typed cryptographic operations; the work on $D\pi$ has produced fairly sophisticated type systems [5,6] to control the access to the resources advertised at the different locations of a distributed system. None of these systems, however, addresses the kind of discretionary policies we have considered here. More precisely, in $D\pi$, resources are created at a specific, unique, type and then delivered to different parties at different (super)types on different channels. Unlike in our system, however, the 'delivery policy' of a value is not described (nor prescribed) by its type. As a consequence, the only guarantee offered by the the type system is that names delivered at type *T* will be received, and re-transmitted freely, at super-types of *T. As* we have illustrated, our types may be employed to specify, and enforce, much more expressive policies.

Type systems have also been proposed to control implicit information flows determined by the behaviour of system components (see [9,14,8,18] among others). These type systems trace the causality relations between computational steps in order to detect covert channels. We follow a different approach to express and verify the delivery of (and the access to) the system resources.

Our approach is also related to the large body of existing work on security automata. In fact, the delivery policies we express in our type system could equivalently be described as finite-state automata whose states are structural types and edges are labelled with groups. On the other hand, security automata have traditionally been employed to provide for run-time system monitoring [17] rather than as a basis for the development of static, type-based, security analyses. More interestingly, the possibility to structure our types as automata (and similarly, as XML-like regular expressions) should lead to the development of efficient algorithms for (sub-) type checking. We leave this as a subject of future work.

The use of session types [7] to express and gain control over the different steps of communication protocols shares motivations with our approach. Session types provide

mechanisms to regulate the sequence of events occurring in a two-parties interaction session. Specifically, a session type prescribes what type of values may legally be passed on a given channel, and in which order. On the other hand, our delivery types prescribe how values can be exchanges among different principals. In both approaches, the types employed are finite-state automata: however the information encoded by the states is different and, as we just argued, largely complementary.

References

- 1. M. Abadi and A. D. Gordon. Reasoning about cryptographic protocols in the Spi calculus. In *CONCUR'97*, volume 1243 of *LNCS*, pages 59–73,1997.
- L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. In *Proceedings of CONCUR*, number 1877 in LNCS, pages 365–379. Springer-Verlag, 2000.
- T. Chothia, D. Duggan, and J. Vitek. Type-based distributed access control. In CSFW 2003, pages 170–184. IEEE, 2003.
- 4. M. Hennessy and J. Riely. Information flow vs resource access in the asynchronous πcalculus. *ACM TOPLAS*, 24(5):566–591, 2002.
- 5. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *I&C*, 173:82–120, 2002.
- Matthew Hennessy, Julian Rathke, and Nobuko Yoshida. SafeDpi: a language for controlling mobile code. In *FoSSaCS'04*, number 2987 in LNCS, pages 241–256, 2004.
- K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP* '98, number 1381 in LNCS, pages 122–138. Springer-Verlag, 1998.
- K. Honda, V.T. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In ESOP '00, volume 1782 of LNCS, pages 180–199. Springer-Verlag, 2000.
- 9. Naoki Kobayashi. Type-based information flow analysis for the pi-calculus. Technical Report TR03-0007, Dept. of Computer Science, Tokyo Institute of Technology, 2003.
- 10. B.W. Lampson. Protection. ACM Operating Systems Rev., 8(1):18–24, Jan. 1974.
- C.J. McCollum, J. R. Messing, and L. Notargiacomo. Beyond the pale of mac and dac defining new forms of access control. In *Proc. of IEEE Symposium on Security and Privacy*, pages 190–200, 1990.
- Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol, (4):410–442, 2000.
- 13. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5), 1996.
- Franois Pottier. A simple view of type-secure information flow in the π-calculus. In Proceedings of the 15th IEEE Computer Security Foundations Workshop, pages 320–330, 2002.
- P. Samarati and S. De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In R. Focardi and R. Gorrieri, editors, *FOSAD 2002*, number 2171 in LNCS. Springer-Verlag, 2002.
- Ravi S. Sandhu and Qamar Munawer. How to do discretionary access control using roles. In ACM Workshop on Role-Based Access Control, pages 47–54, 1998.
- 17. Fred B. Schneider. Enforceable security policies. ACM Trans. Inf. Syst. Secur., 3(1):30–50, 2000.
- P. Sewell and J. Vitek. Secure composition of untrusted code: Boxmpi, wrappers and causality types. *Journal of Computer Security*, 11(2):135–188, 2003.

Elimination of Quantifiers and Undecidability in Spatial Logics for Concurrency

Luís Caires¹ and Étienne Lozes²

¹ Departamento de Informática FCT/UNL, Lisboa, Portugal ² LIP, École Normal Supérieure de Lyon, France

Abstract. Aiming at a deeper understanding of the essence of spatial logics for concurrency, we study a minimal spatial logic without quantifiers or any operators talking about names. The logic just includes the basic spatial operators void, composition and its adjunct, and the next step modality; for the model we consider a tiny fragment of CCS. We show that this core logic can already encode its own extension with quantifiers, and modalities for actions. From this result, we derive several consequences. Firstly, we establish the intensionality of the logic, we characterize the equivalence it induces on processes, and we derive characteristic formulas. Secondly, we show that, unlike in static spatial logics, the composition adjunct adds to the expressiveness of the logic, so that adjunct elimination is not possible for dynamic spatial logics, even quantifier-free. Finally, we prove that both model-checking and satisfiability problems are undecidable in our logic. We also conclude that our results extend to other calculi, namely the π -calculus and the ambient calculus.

Introduction

The introduction of spatial logics in concurrency has been motivated by a recent shift of focus from monolithic concurrent systems towards distributed computing systems. Such systems are by nature both concurrent and spatially distributed, in the sense that they are composed from a number of separate and independently observable units of behavior and computation. The central idea behind spatial logics is that for specificying distributed computations there is a need to talk in a precise way not just about pure behaviors, as is the case with traditional logics for concurrency, but about a richer model able to represent computation in a space. Such an increased degree of expressiveness is necessary if we want to specify with and reason about notions like locations, resources, independence, distribution, connectivity, and freshness. Spatial logics have been proposed for π -calculi [4, 3], and for the ambient calculus [11,10]. Spatial logics for manipulating and querying semi-structured data have also been developed [9,8]. Closely related are the separation logics [20,19], introduced with the aim of supporting local reasoning about imperative programs.

The simplest spatial logic for concurrency, we may argue, is the one obtained by adding to boolean logic the very basic spatial connectives, namely void (0), composition (- | -) and its logical adjunct $(- \triangleright -)$, and then the dynamic modality next step $(\diamondsuit -)$. This logic, based essentially on spatial observations, will be referred here by \mathcal{L}_{spat} .

The basic spatial connectives can be used to specify the distribution of processes, 0 specifies the empty system (not to be confused with the inactive system), and $\mathcal{A} \mid \mathcal{B}$ specifies the systems that can be partitioned in two parts, one satisfying A and the other satisfying B. For the adjunct, $A \triangleright B$ is satisfied by those processes that, whenever composed with a process satisfying A, are guaranteed to satisfy \mathcal{B} . A simple example of a property combining spatial and dynamic operators is the one expressed by the formula $(\neg 0 | \neg 0) \land \Diamond 0$; it specifies those processes that have (at least) two separate components and may reduce to the empty system. Adjuncts allow the specification of contextual properties, e.g., consider the formula $1 \land (1 \triangleright \Diamond 0)$, that uses the existential version of the adjunct defined $\mathcal{A} \triangleright \mathcal{B} \triangleq \neg (\mathcal{A} \triangleright \neg \mathcal{B})$. This formula specifies the single-thread processes that can be composed with some other process to yield a system than may evolve to the empty system, after a single reduction step. Adjunct-free spatial logics with behavioral observations (e.g., [2]) are also able to render some kinds of contextual properties. For example, the property just presented can be expressed by the formula $1 \land \exists x. \langle x \rangle 0$, using an action modality. Thus, one of the motivations for this work is to get a deeper understanding about the relative expressiveness of these approaches.

For the sake of simplicity and generality, we interpret \mathcal{L}_{spat} in a rather small fragment of choice-free CCS. This calculus turns out to conveniently abstract the kind of concurrent behavior present in both π - and ambient calculi, in the broad sense that interactions are local, and triggered by the presence of named capabilities.

At first, \mathcal{L}_{spat} seems quite weak, as far as expressiveness is concerned, when compared to other spatial logics. For instance, it provides no constructs referring to names or actions (like *e.g.*, the action modality $\langle n \rangle \mathcal{A}$ of behavioral logics, or the ambient match construction $n[\mathcal{A}]$ in the ambient logic), therefore formulas of \mathcal{L}_{spat} are always closed. As a consequence, satisfaction of \mathcal{L}_{spat} formulas is invariant under swapping of any pair of actions in processes (a property usually called equivariance) because formulas cannot single out specific actions or names. Still, due to the presence of the \Diamond operator, the logic is able to make some distinctions between actions, and substitution of actions does not in general preserve satisfaction. For instance, let $P \triangleq \alpha \mid \overline{\beta}$. Then $P \models \neg \Diamond \top$ for $\beta \neq \alpha$, but $P\{\beta \leftarrow \alpha\} \not\models \neg \Diamond \top$. These considerations lead to the general question of what is the largest relation between processes which are indistinguishable by the logical equivalence: answering this question crucially contributes to our understanding of the spatial model induced on processes by the simplest combination of logical observations.

However, this question turns out to be a rather difficult one to answer, due to the presence of the composition adjunct operator \triangleright . The adjunct is quite powerful, allowing the logic to perform quite strong observations on processes. With adjunct, validity can be internally defined [11] (thus validity-checking is

subsumed by model-checking), and use certain forms of specification akin to a comprehension principle (for example, we may specify the set of all processes that have an even number of parallel components). The study of expressiveness for spatial logics usually goes through the definition of an adequate spatial bisimilarity \approx along the lines of [16]. Then, establishing the congruence of \approx is key to ensure correctness of \approx , so that from $P \approx Q$ we conclude $P \mid R \approx Q \mid R$. For our logic however, such property does not hold, due to equivariance. For instance, the processes α . **0** and β . **0** are logically equivalent, but α . **0** $\mid \overline{\alpha}$. **0** and β . **0** $\mid \overline{\alpha}$. **0** are not. Hence, this approach does not work well in this setting.

Despite many works about decidability of spatial logics, the question of model-checking spatial logics for concurrency with adjunct has not been fully settled. Results are known for some cases, where the logic includes just \triangleright or \diamondsuit [11, 2], but there seems to be no work about the interesting combination of \triangleright and \diamondsuit , as far as decidability is concerned. However, we believe that this issue lies at the heart and novelty of a purely spatial approach to verification of distributed systems. On the one hand, image-finiteness of the reduction relation gives a model-checking algorithm for adjunct-free logics [2]. On the other hand, in the absence of name quantifiers and name revelation it is also known that static fragments are decidable [6], so there could be some hope in obtaining decidability of model-checking the whole of \mathcal{L}_{spat} .

We may answer these questions considering the extension \mathcal{L}_{mod} of \mathcal{L}_{spat} with the existential quantifier and quantified action modalities; for \mathcal{L}_{mod} , logical equivalence is much clearly intensional, and one may adapt the results of [13] to derive the undecidability of model-checking. But even if \triangleright induces undecidability, we may ask the question of its actual contribution to the expressiveness of the logic. In previous work [18], Lozes has shown that in static spatial logics, that is spatial logics without quantifiers and dynamic operators, the adjunct connective can be eliminated in behalf of the remaining connectives, in the sense that for any formula of such a logic there is a (possibly hard to find) logically equivalent adjunct-free formula. An interesting question is then whether something similar happens in \mathcal{L}_{mod} : we could possibly think that the expressive power of the adjunct could somehow be recovered by the presence of action modalities, given that both kinds of constructs allow some contextual observations to be made.

So \mathcal{L}_{mod} and \mathcal{L}_{spat} seem quite different as far as expressive power is concerned. The first one seems clearly intensional (in the technical sense that logical equivalence coincides with structural congruence), and undecidable. But for the second, as discussed above, it would be reasonable to hope for decidability, and expect a separation power coarser than structural congruence. All this turns out not to be the case.

The key result of this paper is that \mathcal{L}_{mod} admits the elimination of quantifiers and action modalities in a precise sense (Theorem 2.1); on the way we also show that equality is internally definable. Building on this surprising result, we then show that \mathcal{L}_{spat} and \mathcal{L}_{mod} have the same separation power (Theorem 3.3), and expressiveness in a certain sense. As a consequence, we also characterize the separation power of \mathcal{L}_{spat} , showing that it coincides with structural congruence modulo permutation of actions (Theorem 3.3). Quantifier elimination is compositional and effective, allowing us to conclude that model-checking of both \mathcal{L}_{spat} and \mathcal{L}_{mod} is undecidable (Theorem 6.1). A counterexample inspired by a suggestion of Yang allows us then to prove that composition adjunct contributes in a non-trivial way to the expressiveness of both logics, thus settling a conjecture formulated in [18] about whether this connective could also be eliminated in spatial logics for concurrency. We conclude with a generalization of these results to π -calculus and Mobile Ambients.

Related Work. Sangiorgi first showed [21] that observation of capabilities in the ambient calculus can be expressed inside spatial logics making use of the \triangleright and \diamondsuit operators. This result has since then been generalized to other calculi [4,17]. However, in all such encodings, the use of quantifiers, and references to (some times fresh) names using the revelation connective seems to be essential. From this point of view, our work gives a tighter bound on the level of expressiveness really needed to embed action modalities, since it does not use operators beyond those expected in every pure spatial logic. A related effort addressing minimality is being developed by Hirschkoff, characterizing π -calculus behavioral equivalences with a logic with composition adjunct [15].

Adjunct elimination for a static spatial logic was first proved in [18], where a counterexample to adjunct elimination in the presence of quantifiers was also presented. However, the particular counterexample given there makes an essential use of name revelation, and thus only applies to calculi with hidden names and related logical connectives. The counterexample presented here is much more general to spatial logics, since it does not rely on such constructs.

Concerning decidability and model-checking of spatial logics, decidability of model-checking for the adjunct-free ambient logic against the replication free calculus was settled by Cardelli and Gordon in [11]. Validity and model-checking of ambient calculus against spatial logics with existential quantifiers was shown undecidable by Charatonik and Talbot [13]. The same authors also extended the results of [11] to logics with constructs for restricted names, and then with Gordon to the finite-control ambient-calculus [12]. Model-checking the π -calculus against full adjunct-free spatial logic with behavioral modalities, hidden and fresh name quantifiers, and recursion was shown to be decidable in [2]. Decidability of validity in a static spatial logic for trees with adjunct was first shown by Calcagno, Cardelli and Gordon in [6], building on techniques of [7]. More recently, Conforti and Ghelli proved that similar results do not hold in logics with operators for restricted names [14].

To our best knowledge, no results about expressiveness and decidability of dynamic spatial logics so crisp as ours have been presented, in the sense that they apply to a minimal spatial logic for concurrency, and focus on the crucial combination of the composition adjunct with the dynamic modality. The elimination of quantifiers (although not of variables, as we also achieve here) is an important topic of interest in classical logic, related to decidability and complexity issues (*e.g.*, see [1]). However, we believe our work lies completely out

of this scope, as on the contrary we derive undecidability of our logic from the elimination of quantifiers.

1 Preliminaries

In this section, we introduce the process calculus and spatial logics considered in this work. For the process calculus, we pick a fairly small fragment of CCS.

Definition 1.1. Assume given an infinite set A of actions, ranged over by α, β . Processes are defined by the grammar: $P,Q,R ::= \mathbf{0} | P | Q | \alpha . P$.

Actions are given in pairs of distinct (co)actions, characterized by the involution **co**: $A \rightarrow A$ sending α into $\overline{\alpha}$, and such that $\overline{\overline{\alpha}} = \alpha$. The relation of *structural congruence* is defined as the least congruence \equiv on processes such that $P \mid \mathbf{0} \equiv P$, $P \mid Q \equiv Q \mid P$, and $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$. Structural congruence represents identity of the spatial structure of processes. Dynamics of processes is captured by labeled transitions.

Definition 1.2. Given the set $L \triangleq \{\tau\} \cup A$ of labels, the relation of labeled transition is defined by the rules

 $\alpha. P \xrightarrow{\alpha} P \quad P \xrightarrow{\ell} P' \Rightarrow P \mid Q \xrightarrow{\ell} P' \mid Q \quad P \xrightarrow{\alpha} P', Q \xrightarrow{\overline{\alpha}} Q' \Rightarrow P \mid Q \xrightarrow{\tau} P' \mid Q'$

Notice that $\xrightarrow{\alpha}$ is closed under \equiv , and that $\xrightarrow{\tau}$ corresponds to the usual relation of *reduction*, noted \longrightarrow . We define the *depth* of a process *P* (maximal nesting of actions in a process *P*) by letting ds(0) = 0, $ds(\alpha, P) = 1 + ds(P)$, and $ds(P \mid Q) = \max(ds(P), ds(Q))$. Let M_K denote the set of all processes whose depth does not exceed $K: M_K \triangleq \{P \mid ds(P) \leq K\}$. Then $M_{\infty} \triangleq \bigcup_{k \in \mathbb{N}} M_k$ coincides with the set of all processes. We also define the projection (by truncation) $\pi_k: M_{\infty} \to M_k$, by induction on k by letting $\pi_0(P) = 0$, $\pi_{k+1}(0) \triangleq 0$, $\pi_k(P \mid Q) \triangleq \pi_k(P) \mid \pi_k(Q)$, and $\pi_{k+1}(\alpha, P) \triangleq \alpha, \pi_k(P)$.

Having defined the intended process model, we turn to logics. The logic we consider includes the basic spatial operators found in all spatial logics namely: the composition operator |, the void operator 0, and the composition adjunct operator \triangleright (guarantee). To these connectives, we add the temporal operator \diamondsuit (next step), to capture the dynamic behavior of processes. These operators may be considered the core connectives for spatial logics for concurrency. We then consider the extension of the core with modalities for actions (*cf.* Hennessy-Milner logic), and quantifiers ranging over actions.

Definition 1.3. Given an infinite set X of variables, $(x, y \in X)$ formulas are given by:

$$\begin{array}{c|c} \mathcal{A}, \mathcal{B} ::= \mathcal{A} \land \mathcal{B} \mid \mathcal{A} \mid \mathcal{B} \mid \neg \mathcal{A} \mid \mathcal{A} \triangleright \mathcal{B} \mid 0 \mid \Diamond \mathcal{A} & (\mathcal{L}_{spat}) \\ \mid \langle x \rangle \mathcal{A} \mid \langle \overline{x} \rangle \mathcal{A} \mid \exists x. \mathcal{A} & (\mathcal{L}_{mod}) \end{array}$$

$$\begin{array}{l} P, v \models_{M} \neg \mathcal{A} & \text{if not } P, v \models_{M} \mathcal{A} \\ P, v \models_{M} \mathcal{A} \land \mathcal{B} & \text{if } P, v \models_{M} \mathcal{A} \text{ and } P, v \models_{M} \mathcal{B} \\ P, v \models_{M} 0 & \text{if } P \equiv 0 \\ P, v \models_{M} \mathcal{A} \mid \mathcal{B} & \text{if } \exists Q, R. \ P \equiv Q \mid R \text{ and } Q, v \models_{M} \mathcal{A} \text{ and } R, v \models_{M} \mathcal{B} \\ P, v \models_{M} \mathcal{A} \triangleright \mathcal{B} & \text{if } \forall Q \in M, \ Q, v \models_{M} \mathcal{A} \text{ implies } P \mid Q, v \models_{M} \mathcal{B} \\ P, v \models_{M} \exists x. \mathcal{A} & \text{if } \exists \alpha \in \mathcal{A}. \ P, (v \{x \leftarrow \alpha\}) \models_{M} \mathcal{A} \\ P, v \models_{M} \Diamond \mathcal{A} & \text{if } \exists P'. \ P \longrightarrow P' \text{ and } P', v \models_{M} \mathcal{A} \\ P, v \models_{M} \langle x \rangle \mathcal{A} & \text{if } \exists P'. \ P \xrightarrow{\overline{v(x)}} P' \text{ and } P', v \models_{M} \mathcal{A} \\ P, v \models_{M} \langle \overline{x} \rangle \mathcal{A} & \text{if } \exists P'. \ P \xrightarrow{\overline{v(x)}} P' \text{ and } P', v \models_{M} \mathcal{A} \end{array}$$

Fig. 1. Semantics of formulas

We write \mathcal{L}_{spat} for the set of formulas in the pure spatial fragment, and \mathcal{L}_{mod} for the set of all formulas. *Free variables* of formulas are defined as usual; we say a formula is *closed* if it has no free variables. Semantics is defined in Fig. 1 by a relation of satisfaction. *Satisfaction* is expressed by $P, v \models_M \mathcal{A}$ where P is a process, M is a set of processes, \mathcal{A} a formula, and v is a valuation for the free variables of \mathcal{A} . A valuation is a mapping from a finite subset of X to A. For any valuation v, we write $v\{x \leftarrow \alpha\}$ for the valuation v' such that $v'(x) = \alpha$, and v'(y) = v(y) if $y \neq x$. By \emptyset we denote the empty valuation. Notice that this definition of satisfaction matches the usual one except for the presence of the index M, which specifies the range of quantification for interpreting the adjunct (see clause for \triangleright). This generalization is only a convenience for our technical development; it is clear that $\models_{M_{\infty}}$ corresponds to the standard non-relativized relation of satisfaction. So, we abbreviate $P, \emptyset \models_M \mathcal{A}$ by $P, v \models \mathcal{A}$, moreover, when the formula \mathcal{A} is closed we abbreviate $P, \emptyset \models_M \mathcal{A}$ by $P \models_M \mathcal{A}$. By default, the set of processes M is M_{∞} , so that we may abbreviate $P \models_M \mathcal{A}$ for $P \models_{M_{\infty}} \mathcal{A}$.

An action permutation is a bijection $\sigma : A \rightarrow A$ such that $\sigma(\overline{\alpha}) = \overline{\sigma(\alpha)}$. We write $\{\alpha \leftrightarrow \beta\}$ for the action permutation that swaps α and β . Satisfaction verifies the fundamental property of equivariance, which in our present setting is formulated as follows.

Definition 1.4. Let \equiv_s be the binary relation on processes defined by $P \equiv_s Q$ if and only if there is an action permutation σ such that $P \equiv \sigma(Q)$.

Proposition 1.5 (Equivariance). Let $P, v \models_M A$. For every action permutation σ , if $P \equiv \sigma(Q)$ then $Q, \sigma(v) \models_M A$.

We frequently refer to equivalence $=_{L}$ induced on processes induced by the logic L (where L is either \mathcal{L}_{spat} or \mathcal{L}_{mod}). The relation $=_{L}$ is defined by setting $P=_{L}Q$ if for all closed formulas \mathcal{A} , we have P, $\emptyset \models \mathcal{A}$ if and only if $Q, \emptyset \models \mathcal{A}$.

Besides the basic stock of primitive connectives, we also use a few derived ones: we list their definition and formal meaning in Fig. 2. By w(A) we denote

$\bot \triangleq \neg \top$
$\mathcal{A} \Rightarrow \mathcal{B} \triangleq \neg \mathcal{A} \lor \mathcal{B}$
$\mathcal{A} \parallel \mathcal{B} \triangleq \neg (\neg \mathcal{A} \mid \neg \mathcal{B})$
$\mathcal{A}^{\forall} \triangleq \mathcal{A} \parallel \perp$
$\mathcal{A}^{\vdash} \triangleq (\neg \mathcal{A}) \triangleright \bot$
$(x) \Rightarrow \langle 0 \rangle^{\vdash}$ $x = \overline{y} \triangleq ((\langle x \rangle 0 \mid \langle y \rangle 0) \Rightarrow \langle 0 \rangle^{\vdash}$
always
never
$P, v \models_M \mathcal{A} \text{ or } P, v \models_M \mathcal{B}$
$P, v \models_M \mathcal{A} \text{ implies } P, v \models_M \mathcal{B}$
$\forall \alpha \in A. \ P, (v\{x \leftarrow \alpha\}) \models_M \mathcal{A}$
$\forall Q, R. \ P \equiv Q \mid R \ implies \ Q, v \models_M \mathcal{A} \ \text{or} \ R, v \models_M \mathcal{B}$
$\exists Q \in M. \ Q \models_M \mathcal{A} and P \mid Q \models_M \mathcal{B}$
$\forall Q, R. \ P \equiv Q \mid R \ implies \ Q, v \models_M \mathcal{A}$
$\exists Q, R. \ P \equiv Q \mid R \text{ and } Q, v \models_M \mathcal{A}$
$\forall Q \in M. \ Q, v \models_M \mathcal{A}$
$v(x) = v(y) (ext{assuming } M_1 \subseteq M)$
$v(x) = \overline{v(y)}$ (assuming $M_1 \subseteq M$)

Fig. 2. Definition and semantics of derived operators.

the maximal level of nesting of composition | in the formula \mathcal{A} , and by $ds(\mathcal{A})$ the maximal nesting of dynamic modalities in the formula \mathcal{A} , defined by

$$\begin{split} w(0) &= 0 \\ w(\mathcal{A} \land \mathcal{B}) &= w(\mathcal{A} \triangleright \mathcal{B}) = \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= 1 + \max \left(w(\mathcal{A}), w(\mathcal{B}) \right) \\ w(\mathcal{A} \mid \mathcal{B}) &= w(\mathcal{A} \mid \mathcal{B}) \\ w(\mathcal{A} \mid \mathcal{A}) &= w(\mathcal{A}) \\ w(\mathcal{A} \mid \mathcal{A}) \\ w(\mathcal{A} \mid \mathcal{A}) &= w(\mathcal{A}) \\ w(\mathcal{A} \mid \mathcal{A}) \\ w(\mathcal{A} \mid \mathcal{A}) &= w(\mathcal{A}) \\ w(\mathcal{A} \mid \mathcal{A}) \\ w(\mathcal{A} \mid \mathcal{A}) &= w(\mathcal{A}) \\ w(\mathcal{A} \mid \mathcal{A}) \\ w(\mathcal{A}$$

It is easy to see that a formula \mathcal{A} cannot inspect the part of the process that lies deeper than the depth of \mathcal{A} . As a consequence, the restriction to M_k of the denotation of a formula of depth k completely characterizes its denotation, in the precise sense of:

Proposition 1.6 (Depth Finiteness). For all formulas $\mathcal{A} \in \mathcal{L}_{mod}$, for all $k > ds(\mathcal{A})$, for all processes P, and for all valuations v,

$$P, v \models_{M_{\infty}} \mathcal{A}$$
 if and only if $\pi_k(P), v \models_{M_{\infty}} \mathcal{A}$ if and only if $\pi_k(P), v \models_{M_k} \mathcal{A}$.

The following notation will be used. The process $P_1 | \ldots | P_n$ is abbreviated by $\prod_{i=1...n} P_i$, and by P^n we denote the process $\prod_{i=1...n} P$. In the same way, we abbreviate the formula $\mathcal{A}_1 | \ldots | \mathcal{A}_n$ by $\prod_{i=1...n} \mathcal{A}_i$, and \mathcal{A}^n then denotes $\prod_{i=1...n} \mathcal{A}$.

2 Elimination of Quantifiers and Action Modalities

In this section we prove that, quite surprisingly, the logic \mathcal{L}_{mod} , which contains quantifiers and variables, can be embedded into the core logic \mathcal{L}_{spat} , which does not seem to contain related constructs, in the sense of the following main result:

Theorem 2.1. For any closed formula $\mathcal{A} \in \mathcal{L}_{mod}$ and any natural number $K > ds(\mathcal{A})$, we can effectively construct a formula $[\![\mathcal{A}]\!]_K \in \mathcal{L}_{spat}$ such that for all processes P:

 $P \models \mathcal{A}$ if and only if $\pi_K(P) \models \llbracket \mathcal{A} \rrbracket_K$

Notice that this result does not state that \mathcal{L}_{spat} and \mathcal{L}_{mod} have the same expressiveness in the usual sense, however, we should note that the denotation of a formula \mathcal{A} is completely characterized by its denotation on some subset of the models M_k , in the sense of Proposition 1.6. Hence, the denotation of $[\mathcal{A}]_K$ completely characterizes the denotation of \mathcal{A} ; this close correspondence will be enough to show the undecidability and separability of \mathcal{L}_{spat} , and independence of the composition adjunct.

The proof of Theorem 2.1 requires considerable build up. In particular, we need to define \mathcal{L}_{spat} formulas to characterize processes of several quite specific forms, to be used for various purposes in our encoding of \mathcal{A} into $[\![\mathcal{A}]\!]_K$. This exercise turns out to be quite interesting: by going through it we get a better understanding about what can be expressed in \mathcal{L}_{spat} , in a sometimes not really obvious way.

We want to reduce a satisfaction judgment $P, v \models_{M_K} A$, where A is any \mathcal{L}_{mod} formula, into a satisfaction judgment for a formula $[A]_K$ of \mathcal{L}_{spat} that neither contains quantifiers, nor action modalities (and thus no occurrences of variables whatsoever). The key idea is to represent the valuation v appearing in $P, v \models_{M_K} \mathcal{A}$ by a certain process $val(e, v, w)_K$, to be composed with the process P being tested for satisfaction. With this trick, the introduction of the valuation entry for x introduced in the valuation clause for $\exists x. A$ can be mimicked by the introduction of a process using ▶; action modalities are then interpreted by detecting interactions between the process P being tested and the valuation process $val(e, v, w)_K$. More concretely, we encode the pair P, v by a process of the form $P \mid val(e, \nu, w)_K$, where $val(e, \nu, w)_K$ encodes the valuation, and $\nu \circ e = v$ is a decomposition of the valuation v into certain maps $e : X \rightarrow v$ N and $\nu : \mathbb{N} \to \mathbf{A}$, respectively called *environment* and *naming*, and w is a natural number. The role of these data will be explained below. The encoding of valuations makes use of the notion of *row* process. A row process $row(n, \alpha)$ is a sequential process of the form α . α ... α . **0**, where the action α occurs precisely *n* times (so that $ds(row(n, \alpha)) = n$). This process is interesting since it can be characterized logically, and we will use rows to represent bindings between variables (represented by rows of different length) and actions α . Moreover, by imposing a bound K on the depth of the process P one considers, we can easily separate the valuation part from the process that represents the "real" model, in the "soup" $P \mid \mathbf{val}(e, \nu, w)_K$.

We start by introducing formulas whose models are precisely the sequential threads with a given number of actions, in the way we also define the derived modality ?. A.

We have

Lemma 2.2. For all processes P, and M such that $M_1 \subseteq M$

$P \models_M 1$	$i\!f\!f$	$\exists \alpha \in A. \exists Q. P \equiv \alpha. Q$
$P \models_M ?. \mathcal{A}$	$i\!f\!f$	$\exists \alpha \in A. \exists Q. \ P \equiv \alpha. Q \ and \ Q \models \mathcal{A}$
$P \models_M Thread(1)$	iff	$\exists \alpha \in A. \ P \equiv \alpha. 0$
$P \models_M Thread(k)$	$i\!f\!f$	$\exists \alpha_1 \in A, \ldots, \exists \alpha_k \in A, P \equiv \alpha_1, \cdots, \alpha_k, 0$

We now give (for each $k \ge 0$) a formula \mathcal{M}_k that characterizes the model M_k , that is, such that we have $P \models \mathcal{M}_k$ if and only if $P \in M_k$.

$$\mathcal{M}_0 \triangleq 0 \qquad \mathcal{M}_{k+1} \triangleq (1 \Rightarrow ?. \mathcal{M}_k)^{\forall}$$

Using the \diamond modality as an equality tester, we define a formula Equals(k) that is satisfied by the processes which belong to M_k , and are compositions of guarded processes all with the *same* first action. We may then specify rows using appropriate formulas

 $\begin{array}{ll} \mathsf{Equals}(k) & \triangleq \mathcal{M}_k \land (\mathsf{Thread}(k+1) \blacktriangleright \left((\mathsf{Thread}(k+1) \mid 1) \Rightarrow \Diamond \top \right)^{\forall} \right) \\ \mathsf{RowCol}(0) & \triangleq 0 \\ \mathsf{RowCol}(n+1) \triangleq \left(\mathsf{Thread}(n+1) \mid \mathsf{Equals}(1) \right) \land \Diamond \mathsf{RowCol}(n) \\ \mathsf{Row}(n) & \triangleq \mathsf{Thread}(n) \land (\top \blacktriangleright \mathsf{RowCol}(n)) \end{array}$

We now prove

Lemma 2.3. For all k, and process P, we have:

 $\begin{array}{lll} P \models \mathcal{M}_k & \textit{iff} & P \in M_k \\ P \models \mathsf{Equals}(k) & \textit{iff} & P \in M_k \textit{ and } \exists \alpha \in \mathsf{A}. \exists n \ge 0. \\ & \exists P_1, \dots, P_n. & P \equiv \alpha. P_1 \mid \dots \mid \alpha. P_n \\ P \models \mathsf{Row}(k) & \textit{iff} & \exists \alpha \in \mathsf{A}. & P \equiv \mathbf{row}(k, \alpha) \end{array}$

We can now explain our encoding of a valuation v into a certain process. First, we decompose v into two functions ν and e such that $v = \nu \circ e$. An *environment* e is a partial injective function from variables to naturals. When the translation process crosses a $\exists x$ construction, we want to allocate a fresh number to x of the natural numbers; to do so, we note by $e\{x \leftarrow n\}$ the extension of e with $x \mapsto n$, and |e| is the maximal value of e, that, is the number of variables already allocated. A *naming* ν is a function from $[1, \ldots, n]$ to A. Notice that the decomposition $v = \nu \circ e$ is not unique, but will be given by the order in which existential quantified variables are introduced in their scopes. For any naming ν and environment *e* the process **val**(*e*, ν , *w*)_{*K*} is

$$\operatorname{val}(e,\nu,w)_K \triangleq \prod_{i=1...|e|} \operatorname{row}(K+i,\nu_i)^{2^{w}}$$

The parameter w specifies the number of rows of the appropriate length that are needed to represent the environment entry for a variable x, and is related to the number of occurrences of | in the source formulas. Since interpreting | also splits the (encoding of the) valuation, we have to provide enough copies $(2^w,$ where w is related to $w(\mathcal{A})$). Note that we can always filter out any undesirable interference of $\mathbf{val}(e, \nu, w)_K$ with the parallel process P, since for any labeledtransition reduct Q of $\mathbf{val}(e, \nu, w)_K$, Q is not an environment since it does not have the right number of rows for each depth. Likewise, for any namings ν, ν', ν'' , we have $\mathbf{val}(e, \nu, w+1)_K \equiv \mathbf{val}(e, \nu', w)_K | \mathbf{val}(e, \nu'', w)_K$ if and only if $\nu = \nu' =$ ν'' . Using already defined properties, we set

$$\mathsf{Val}(e,w)_K \triangleq \prod_{i=1...|e|} (\mathsf{Row}(K+i)^{2^w} \land \mathsf{Equals}(K+i))$$

 $\mathsf{ProcVal}(e,w)_K \triangleq \mathcal{M}_K | \mathsf{Val}(e,w)_K$

Lemma 2.4. For any process P, environment e and naturals $K, w \ge 1$

The formula $ProcVal(e, w)_K$ specifies a pair process-valuation, where the process belongs to M_K . Now we introduce formulas to match specific entries of the (encoding of the) valuation: selection of the action α associated to the variable x is achieved by filtering the set of row processes of depth e(x).

$$\begin{array}{rcl} \mathsf{XRow}(x,e)_K &\triangleq & \mathsf{Row}(K+e(x)) \\ \mathsf{Used}\mathsf{XRow}(x,e)_K &\triangleq & \mathsf{Row}(K+e(x)-1) \\ \mathsf{Env}\mathsf{X}(x,e,w)_K &\triangleq & \mathsf{Equals}(K+\mid e\mid) \land (\mathsf{XRow}(x,e)_K)^{2^u} \end{array}$$

 $XRow(x, e)_K$ allows us to select one of the rows that represents the environment entry of the variable x. Used $XRow(x, e)_K$ checks that such a row has lost an action prefix (after a reduction step takes place). Env $X(x, e, w)_K$ matches all the rows that encode the environment entry for the variable x. To encode the modality $\langle x \rangle A$ we need to check for the presence of the complementary of the action v(x). To this end, we specify a row longer than any other (with Test(e)), and then check (using \Diamond) that it may react with some row of depth e(x) (with UsedTest(e)). Let then:

$$\begin{array}{rcl} \mathsf{Test}(e)_K &\triangleq & \mathsf{Row}(\mid e \mid +K+2) \\ \mathsf{UsedTest}(e)_K &\triangleq & \mathsf{Row}(\mid e \mid +K+1) \\ \mathsf{TestMatchesX}(x,e,w)_K &\triangleq & (\mathsf{Test}(e)_K \mid \mathsf{EnvX}(x,e,w)_K) \land \Diamond \top \end{array}$$

We are now ready to present our encoding of formulas of \mathcal{L}_{mod} into formulas of \mathcal{L}_{spat} .

$$\begin{split} & \llbracket \mathcal{A} \land \mathcal{B} \rrbracket_{(e,w)} \triangleq \operatorname{ProcVal}(e,w)_K \land \llbracket \mathcal{A} \rrbracket_{(e,w)} \land \llbracket \mathcal{B} \rrbracket_{(e,w)} \\ & \llbracket \neg \mathcal{A} \rrbracket_{(e,w)} \triangleq \operatorname{ProcVal}(e,w)_K \land \neg \llbracket \mathcal{A} \rrbracket_{(e,w)} \\ & \llbracket \partial \rrbracket_{(e,w)} \triangleq \operatorname{ProcVal}(e,w)_K \land \nabla \operatorname{Val}(e,w)_K \\ & \llbracket \mathcal{A} \upharpoonright \mathcal{B} \rrbracket_{(e,w)} \triangleq \operatorname{ProcVal}(e,w)_K \land (\llbracket \mathcal{A} \rrbracket_{(e,w-1)} \mid \llbracket \mathcal{B} \rrbracket_{(e,w-1)}) \\ & \llbracket \mathcal{A} \triangleright \mathcal{B} \rrbracket_{(e,w)} \triangleq \operatorname{ProcVal}(e,w)_K \land (\llbracket \mathcal{A} \rrbracket_{(e,w-1)} \mid \llbracket \mathcal{B} \rrbracket_{(e,w-1)}) \\ & \llbracket \mathcal{A} \triangleright \mathcal{B} \rrbracket_{(e,w)} \triangleq \operatorname{ProcVal}(e,w)_K \land (\llbracket \mathcal{A} \rrbracket_{(e,w)} \vDash \llbracket \mathcal{B} \rrbracket_{(e,w+1)})) \\ & \llbracket \mathcal{A} \triangleright \mathcal{B} \rrbracket_{(e,w)} \triangleq \operatorname{ProcVal}(e,w)_K \land \Diamond \llbracket \mathcal{A} \rrbracket_{(e,w)} \\ & \llbracket \mathcal{A} \rrbracket_{(e,w)} \triangleq \operatorname{ProcVal}(e,w)_K \land (\operatorname{EnvX}(x,e',w)_K \blacktriangleright \llbracket \mathcal{A} \rrbracket_{(e',w)}) \\ & \llbracket \mathcal{A} \rrbracket_{(e,w)} \triangleq \operatorname{ProcVal}(e,w)_K \land (\operatorname{EnvX}(x,e',w)_K \blacktriangleright \llbracket \mathcal{A} \rrbracket_{(e',w)}) \\ & \operatorname{where} e' = e\{x \leftarrow \mid e \mid +1\} \\ & \llbracket \langle x \rangle \mathcal{A} \rrbracket_{(e,w)} \triangleq \operatorname{ProcVal}(e,w)_K \land \\ & \operatorname{Test}(e)_K \blacktriangleright ((\operatorname{TestMatchesX}(x,e,w)_K \mid \top) \land \\ & \land (\operatorname{UsedTest}(e)_K \mid \llbracket \mathcal{A} \rrbracket_{(e,w)})) \\ & \llbracket \langle \overline{x} \rangle \mathcal{A} \rrbracket_{(e,w)} \triangleq \operatorname{ProcVal}(e,w)_K \land \Diamond (\operatorname{UsedXRow}(x,e)_K \mid (\operatorname{XRow}(x,e)_K \blacktriangleright \llbracket \mathcal{A} \rrbracket_{(e,w)})) \end{aligned}$$

Fig. 3. Encoding of \mathcal{L}_{mod} into \mathcal{L}_{spat} .

Definition 2.5. Let $\mathcal{A} \in \mathcal{L}_{mod}$ be a formula, e an environment mapping the free variables of \mathcal{A} , and w, K be integers such that $w > w(\mathcal{A})$, and K > 0. Then, the formula $[\mathcal{A}]_{(e,w)} \in \mathcal{L}_{spat}$ is inductively defined in Fig. 3.

Theorem 2.1 follows from Lemmas 2.2, 2.3, 2.4, and the following general result:

Lemma 2.6 (Correctness of the Encoding). For all processes P, all formulas $A \in \mathcal{L}_{mod}$, all environments e declaring the free variables of A, all integers w > w(A), and all K > 0 we have:

$$P, \emptyset \models_{M_{\infty}} \llbracket \mathcal{A} \rrbracket_{(e,w)} \quad \text{ if and only if } \quad \exists Q \in M_K, \exists \nu. \begin{cases} P \equiv Q \mid \mathbf{val}(e,\nu,w)_K \\ Q, \nu \circ e \models_{M_K} \mathcal{A} \end{cases}$$

Proof. (Sketch, see [5]) By induction on \mathcal{A} . For the connectives of \mathcal{L}_{spat} , the encoding is quite natural: in the case of |, the environment is split in two equal parts, and tested for a sound recombination by $\operatorname{ProcVal}(e, w)_K$. For \triangleright , we must check that the composition of the two environments coming from the left and right of \triangleright is actually an environment. This holds if both environments are defined with the same naming ν . For the case of \diamondsuit , any reduction involving the environment is excluded, because otherwise the resulting environment would be ill-formed. For the other connectives, the encoding also involves our abbreviations: the encoding of the quantification (using \triangleright) over processes that represent environment entries. For action modalities, one checks for interactions between the process and a row corresponding to the selected variable.

We can thus present the proof of Theorem 2.1.

Proof. Let \mathcal{A} be a formula of \mathcal{L}_{mod} . Set $[\![\mathcal{A}]\!]_K = [\![\mathcal{A}]\!]_{(\emptyset,w)}$ for some w greater than the maximal nesting of | connectives in \mathcal{A} . Then $\pi_K(P) \equiv \pi_K(P) |$

val $(\emptyset, \emptyset, w)_K$, so by Lemma 2.6, $\pi_K(P), \emptyset \models_{M_{\infty}} \llbracket \mathcal{A} \rrbracket_K$ if and only if $\pi_K(P), \emptyset \models_{M_K} \mathcal{A}$, which is equivalent to $P, \emptyset \models_{M_{\infty}} \mathcal{A}$ by Proposition 1.6.

3 Separability of \mathcal{L}_{spat}

As a first application of the main Theorem 2.1, we define characteristic formulas and characterize the separation power of the logic \mathcal{L}_{spat} (and thus of \mathcal{L}_{mod}). We conclude that \mathcal{L}_{spat} is able to describe processes quite precisely, just abstracting away from the identity of the particular names used by processes. We start by introducing a characteristic formula C(P) for any process P. For any complementary pair of actions $\{\alpha, \overline{\alpha}\}$ occurring in P, we reserve a specific variable x_a , collected in the set $\{x_{\alpha_1}, \ldots, x_{\alpha_n}\}$.

$$\chi(\mathbf{0}) \triangleq 0 \qquad \chi(\alpha, P) \triangleq 1 \land \langle x_{\alpha} \rangle \chi(P) \chi(\overline{\alpha}, P) \triangleq 1 \land \langle \overline{x_{\alpha}} \rangle \chi(P) \qquad \chi(P \mid Q) \triangleq \chi(P) \mid \chi(Q) C(P) \triangleq [\exists x_{\alpha_{1}} \dots \exists x_{\alpha_{n}} . (\bigwedge_{i \neq j} x_{\alpha_{i}} \neq x_{\alpha_{j}} \land x_{\alpha_{i}} \neq \overline{x_{\alpha_{j}}}) \land \chi(P)]_{K}$$

where K = ds(P). Recall that abbreviations (x = y) and $(x \neq y)$ are defined in Fig.2, and notice that $C(P) \in \mathcal{L}_{spat}$, while $\chi(P) \in \mathcal{L}_{mod}$.

Lemma 3.1. Let $P \in M_K$, let v be the valuation such that $v(x_{\alpha_i}) = \beta_i$, for pairwise distinct actions β_1, \ldots, β_n , and let σ be the action permutation that sends α_i into β_i . Then we have that $Q, v \models_{M_K} \chi(P)$ if and only $Q \equiv \sigma(P)$.

Proof. Induction on P (see [5]).

Lemma 3.2. For all processes Q and P, $Q \models C(P)$ if and only if $Q \equiv_s P$.

Proof. By Lemma 3.1 and Theorem 2.1 (see [5]).

We then conclude:

Theorem 3.3. The following statements are equivalent:

(1) $P =_{\mathcal{L}_{mod}} Q$ (2) $P =_{\mathcal{L}_{spat}} Q$ (3) $Q, \emptyset \models C(P)$ (4) $P \equiv_{s} Q$

Proof. (1) \Rightarrow (2) because $\mathcal{L}_{spat} \subset \mathcal{L}_{mod}$, (2) \Rightarrow (3) since $C(P) \in \mathcal{L}_{spat}$ and $P \models C(P)$, (3) \Rightarrow (4) by Lemma 3.2, and (4) \Rightarrow (1) by Proposition 1.5.

4 Expressiveness of Composition Adjunct

It is known that in static spatial logics, that is spatial logics without quantifiers and dynamic operators, the adjunct connective is not independent of the remaining connectives, and can in fact be eliminated, in the sense that for any formula of such a logic we can find a logically equivalent adjunct-free formula [18]. It is not hard to see that adjunct cannot be dispensed with in \mathcal{L}_{spat} , because without adjunct one is not allowed to distinguish threads of different lenght: if we pick $\mathcal{A} \in \mathcal{L}_{spat} - \{\triangleright\}$, we can verify by an easy induction on \mathcal{A} that $\alpha . 0 \models \mathcal{A}$ if and only if $\alpha . \beta . 0 \models \mathcal{A}$, for all $\alpha, \beta \in A$.

In this section, we prove that the adjunct elimination property does not hold for the spatial logic \mathcal{L}_{mod} . For this, we adapt a scheme suggested by Yang: on the one hand, we define in \mathcal{L}_{mod} a formula that says of a process that its number of toplevel parallel components is even, on the other hand, we show that parity cannot be characterized by adjunct-free formulas. We start by defining a few formulas (where $\Box \mathcal{A} \triangleq \neg \Diamond \neg \mathcal{A}$):

$$\begin{array}{l} \mathsf{Top}(x) \triangleq \langle x \rangle 0 \\ \mathsf{Fam} \quad \triangleq \Box \bot \land \left(1 \Rightarrow \exists x. \, \mathsf{Top}(x) \right)^{\forall} \land \forall x. \, \forall y. \, (\mathsf{Top}(x) \mid \mathsf{Top}(y) \mid \top) \Rightarrow x \neq y) \end{array}$$

We can verify that $P \models \mathsf{Fam}$ if and only if $P \equiv \alpha_1.0 \mid \ldots \mid \alpha_k.0$ for some pairwise distinct k actions $\alpha_1, \ldots, \alpha_k$ such that $P \not\rightarrow \ldots$ We call a process of such a form a *family*. The width of such a family P is defined to be the number w(P) = k of parallel threads in P. Now, we can define a formula Even2 that is satisfied by processes that contain exactly an even number of distinct actions at the second level.

Pair
$$\triangleq 1 \land \exists xyz. \langle x \rangle (\mathsf{Top}(y) \mid \mathsf{Top}(z)) \land (y \neq z)$$
Below $(x) \triangleq 1 \land \exists z. \langle z \rangle \langle x \rangle \top$ Even2 $\triangleq (1 \Rightarrow \mathsf{Pair})^{\forall} \land \forall x. \forall y. (\mathsf{Below}(x) \mid \mathsf{Below}(y) \mid \top) \Rightarrow x \neq y)$

Hence $P \models \text{Even2}$ if and only if $P \equiv \alpha_1$. $(\beta_{1,1}.0 \mid \beta_{1,2}.0) \mid \cdots \mid \alpha_k$. $(\beta_{k,1}.0 \mid \beta_{k,2}.0)$ for some k actions $\alpha_1, \ldots, \alpha_k$, and some pairwise distinct 2k actions $\beta_{1,i}, \ldots, \beta_{k,i}$ for i = 1, 2. Now, if we compose a process P satisfying Fam in parallel with a process Q satisfying Even2, we can check (in $P \mid Q$) that the actions that occur in the toplevel of P are exactly the same that appear in the second level of Q using the formula Same:

Same
$$\triangleq \forall x. (\mathsf{Top}(x)^{\exists} \Leftrightarrow \mathsf{Below}(x)^{\exists})$$

Hence we have the following result

Lemma 4.1. There is a closed formula $\mathsf{Even} \in \mathcal{L}_{mod}$ such that for any process P, we have that $P \models \mathsf{Even}$ if and only if P is a family and w(P) is even.

```
Proof. Let Even \triangleq Fam \land (Even2 \triangleright Same).
```

A key observation is that the formula Even contains an essential use of the composition adjunct operator. In fact, although the properties denoted by the formulas Even2 and Fam can be expressed by appropriate adjunct-free formulas of \mathcal{L}_{spat} , the same situation does not hold for the parity property expressed by Even. In the remainder of this section, we prove that there is no formula of $\mathcal{L}_{mod} - \{\triangleright\}$ able to express the same property. The argument consists in showing

that any family *P* considered in $\mathcal{L}_{mod} - \{\triangleright\}$ admits a saturation level from which it is always possible to add an extra parallel component to it while preserving satisfaction. We first define $sn(\mathcal{A})$ (the *sticks number* of the formula \mathcal{A}) to be the natural number defined by induction on \mathcal{A} as follows:

$$\begin{array}{ll} \operatorname{sn}(\neg \mathcal{A}) & \triangleq \operatorname{sn}(\mathcal{A}) & \operatorname{sn}(\mathcal{A}_1 \land \mathcal{A}_2) \triangleq \max(\operatorname{sn}(\mathcal{A}_1), \operatorname{sn}(\mathcal{A}_2)) \\ \operatorname{sn}(0) & \triangleq 1 & \operatorname{sn}(\mathcal{A}_1 \mid \mathcal{A}_2) \triangleq \operatorname{sn}(\mathcal{A}_1) + \operatorname{sn}(\mathcal{A}_2) \\ \operatorname{sn}(\Diamond \mathcal{A}) & \triangleq 0 & \operatorname{sn}(\langle x \rangle, \mathcal{A}) \triangleq \operatorname{sn}(\mathcal{A}) \\ \operatorname{sn}(\exists x, \mathcal{A}) \triangleq \operatorname{sn}(\mathcal{A}) + 1 & \operatorname{sn}(\langle \overline{x} \rangle, \mathcal{A}) \triangleq \operatorname{sn}(\mathcal{A}) \end{array}$$

Given a family *P* and a valuation *v*, we write $P \setminus v$ for the subfamily of *P* of the actions α that do not appear in the codomain of the valuation *v*. More precisely, we define $P \setminus v \triangleq \prod \{\alpha, \mathbf{0} : P \equiv \alpha, \mathbf{0} \mid Q \text{ and } \alpha, \overline{\alpha} \notin \text{codom}(v) \}$. We then have:

Lemma 4.2. Let P be a family, let v be a valuation $v : X \rightarrow A$, and let $\alpha \in A$ be an action such that $\alpha, \overline{\alpha} \notin \operatorname{codom}(v)$ and $(P \mid \alpha, 0)$ is a family. Then, for any \triangleright -free formula $A \in \mathcal{L}_{mod}$ such that $w(P \setminus v) \geq \operatorname{sn}(A)$ we have

 $P, v \models A$ if and only if $P \mid \alpha.0, v \models A$.

Proof. By induction on \mathcal{A} (see [5]).

Theorem 4.3. There is no closed formula $A \in \mathcal{L}_{mod} - \{\triangleright\}$ that exactly characterizes the set of all families P with w(P) even.

Proof. By contradiction: if \mathcal{A} was a such formula, then we may take a family P and an extended family $P \mid \alpha$ with $w(P) \ge \operatorname{sn}(\mathcal{A})$. Then by previous lemma, $P, \emptyset \models \mathcal{A}$ if and only if $P \mid \alpha, \emptyset \models \mathcal{A}$, which is a contradiction.

We thus conclude that in the logic \mathcal{L}_{mod} the composition adjunct operator is independent of the remaining operators, in particular there are properties expressible with the composition adjunct that cannot be expressed with action modalities and quantifiers.

5 Undecidability

In this section, we show that the validity-, satisfiability- and model-checking problems for the logic \mathcal{L}_{spat} (and hence for \mathcal{L}_{mod}) are all undecidable. These results are a consequence of our embedding of \mathcal{L}_{mod} into \mathcal{L}_{spat} (Theorem 2.1), and of the fact that first-order logic can then be easily encoded into \mathcal{L}_{mod} (along the lines of [13]). The language of first-order logic (FOL) and its semantics is defined as usual:

$$\mathcal{A}, \mathcal{B} ::= \mathcal{A} \wedge \mathcal{B} \mid \neg \mathcal{A} \mid \exists x. \mathcal{A} \mid p(x, y)$$

Formulas of FOL are built from a set *Vars* of individual variables (x, y), and without loss of generality we consider a single binary predicate symbol p. A model for FOL is a pair (D, I) where D is a set of individuals (the domain of the

model), and *I* is a binary relation $I \subseteq D \times D$. For our purposes it is enough to focus on finite models. Satisfaction of a FOL formula by a model is defined using a valuation v that assigns each individual variable an element of *D* as follows:

$$\begin{array}{ll} (D,I) \models_{v} \mathcal{A} \land \mathcal{B} & if \ (D,I) \models_{v} \mathcal{A} \ and \ (D,I) \models_{v} \mathcal{B} \\ (D,I) \models_{v} \neg \mathcal{A} & if \ not \ (D,I) \models_{v} \mathcal{A} \\ (D,I) \models_{v} \exists x. \mathcal{A} & if \ \exists \ d \in D. \ (D,I) \models_{v \{x \leftarrow d\}} \mathcal{A} \\ (D,I) \models_{v} p(x,y) & if \ (v(x), v(y)) \in I \end{array}$$

We now show how to encode any FOL satisfaction judgment $(D, I) \models_v \mathcal{A}$ into a \mathcal{L}_{mod} satisfaction judgment $\mathcal{M}[\![(D, I)]\!], \mathcal{V}[\![v]\!] \models \mathcal{F}[\![A]\!]$, by means of appropriate translations $\mathcal{M}[\![-]\!], \mathcal{V}[\![-]\!]$ and $\mathcal{F}[\![-]\!]$. We pick natural numbers K, E such that K > E > 2. To encode a model (D,I) into a process $\mathcal{M}[\![(D,I)]\!]$, we start by assigning each element $d \in D$ a distinct action $A(d) \in A$, and define $\mathcal{E}[\![d]\!] \triangleq \operatorname{row}(E, A(d))$. The domain $D = \{d_1, \ldots, d_n\}$ is represented by the process $\mathcal{D}[\![D]\!] \triangleq \mathcal{E}[\![d_1]\!] \mid \ldots \mid \mathcal{E}[\![d_k]\!]$. For the interpretation I, we represent each pair $(d, e) \in I$ by the process $\mathcal{T}[\![(d, e)]\!] \triangleq A(d).A(e).0$. We then let $\mathcal{T}[\![I]\!] \triangleq \prod_{(d,e)\in I} \mathcal{T}[\![(d,e)]\!]$ and finally set $\mathcal{M}[\![(D,I)]\!] \triangleq \mathcal{D}[\![D]\!] \mid \mathcal{I}[\![I]\!]$. Notice that, by construction, we always have $\mathcal{M}[\![(D,I)]\!] \in M_K$. Processes encoding our FOL models can be characterized by a formula Model of \mathcal{L}_{spat} as follows:

 $\begin{array}{ll} \mathsf{D}(x) & \triangleq \mathsf{Row}(E) \land \langle x \rangle \top & \mathsf{Diff} & \triangleq \forall x. \forall y. (\langle x \rangle \top \mid \langle y \rangle \top) \Rightarrow x \neq y) \\ \mathsf{Domain} \triangleq \mathsf{Diff} \land (1 \Rightarrow \exists x. \mathsf{D}(x))^{\forall} \mathsf{Cmp} & \triangleq \forall x. \forall y((\langle x \rangle \langle y \rangle 0)^{\exists} \Rightarrow (\mathsf{D}(x)^{\exists} \land \mathsf{D}(y)^{\exists})) \\ \mathsf{Interp} & \triangleq (1 \Rightarrow \mathsf{Thread}(2))^{\forall} & \mathsf{Model} \triangleq \mathcal{M}_K \land \llbracket (\mathsf{Domain} \mid \mathsf{Interp}) \land \mathsf{Cmp} \rrbracket_K \end{array}$

Lemma 5.1. $P \models \text{Model iff there is a finite FOL model (D,I) and <math>\mathcal{M}[[(D,I)]] \equiv P$.

Proof. Interpreting the formula Model (see [5])).

Then, formulas of FOL are encoded into formulas of \mathcal{L}_{mod} as follows

$$\begin{array}{l} \mathcal{F}\llbracket \neg \mathcal{A} \rrbracket \triangleq \neg \mathcal{F}\llbracket \mathcal{A} \rrbracket \qquad \mathcal{F}\llbracket \exists x. \mathcal{A} \rrbracket \triangleq \exists x. (\mathsf{D}(x)^{\exists} \land \mathcal{A}) \\ \mathcal{F}\llbracket \mathcal{A} \land \mathcal{B} \rrbracket \triangleq \mathcal{F}\llbracket \mathcal{A} \rrbracket \land \mathcal{F}\llbracket \mathcal{B} \rrbracket \qquad \mathcal{F}\llbracket p(x, y) \rrbracket \triangleq (1 \land \langle x \rangle \langle y \rangle 0)^{\exists} \end{array}$$

Finally, for valuations we set $\mathcal{V}[v](x) = A(v(x))$. We can prove

Lemma 5.2. Let $v = \{x_1 \mapsto d_1, \ldots, x_k \mapsto d_k\}$ be a valuation for \mathcal{A} . Then we have $(D, I) \models_v \mathcal{A}$ if and only if $\mathcal{M}[[(D, I)]], \mathcal{V}[v]] \models_{M_K} \mathcal{F}[[\mathcal{A}]]$.

Proof. See [5].

Proposition 5.3. Let \mathcal{A} be a closed formula of FOL. Then the formula \mathcal{A} is satisfiable if and only if the \mathcal{L}_{spat} formula Model $\wedge [\![\mathcal{F}[\![\mathcal{A}]]\!]_{K}$ is satisfiable.

Proof. By Lemma 5.2, Lemma 5.1 and Theorem 2.1 (see [5]).

As a corollary of Proposition 5.3, we conclude

Theorem 6.1. The problems of validity-checking, satisfiability-checking, and model-checking of \mathcal{L}_{spat} formulas are all undecidable.

Proof. Follows from Proposition 5.3 and Trakhtenbrot's Theorem [22].

6 Extension to the π -Calculus and Ambients

In this section, we briefly discuss how our results extend to richer models, namely the π -calculus and the ambient calculus. We may pick any of these calculi as models for the core logic \mathcal{L}_{spat} , which is a fragment of both the ambient logic of [11] and the π -calculus logic of [4]. We discuss first the case of the ambient calculus without name restriction, and just with the open capability. In this case, we can show that \mathcal{L}_{spat} can also encode, for processes of bounded depth, its extension with the quantifier $\exists x. \mathcal{A}$, and modalities of the form $\langle \text{open } x \rangle$. \mathcal{A} and $x[\mathcal{A}]$. However, as we might expect, the symmetry between input and output (Theorem 3.3(4)) does not carry over to ambients: for instance, the formula $1 \land \Diamond \top$ may be satisfied by the ambient n[P], but not by the guarded ambient open n. P. For the π -calculus, we may consider the extension of \mathcal{L}_{spat} with the quantifier $\exists x. \mathcal{A}$ and the modalities $\langle x \rangle \mathcal{A}$ and $\langle \overline{x} \rangle \mathcal{A}$, able to observe just the subjects of π -calculus actions. In this case, we may also prove that this extension can be encoded in \mathcal{L}_{spat} for bounded depth processes, as we did for the other cases. From these results, we conclude

Theorem 6.2. The model-checking and validity problems for the π -calculus and the ambient calculus against \mathcal{L}_{spat} are both undecidable.

Proof. See [5].

We should remark that Trakhtenbrot also allows us to conclude that there is no complete proof system for validity of \mathcal{L}_{spat} formulas over any of these calculi.

7 Concluding Remarks

We have studied a core spatial logic for concurrency, aiming at a better understanding of the relative role of the very basic logical operations present in most logics of this family. In particular, we have shown that quantifiers and action modalities can be embedded, and that the composition adjunct plays a key role in the expressiveness of this logic; these results allowed us to also prove its undecidability. Ours results are expected to hold for most process calculi, even in the presence of recursion or replication. In this light, we believe that minimality of \mathcal{L}_{spat} could be established in a precise sense.

The logics \mathcal{L}_{spat} and \mathcal{L}_{mod} have not been shown to have the same expressiveness in the strict technical sense. However, we believe this is the case for their extension with freshness quantifiers and a free name occurrence predicate. Since Theorem 3.3(4) does not hold for calculi with name restriction, an interesting issue is to get a better understanding of the (coarser) spatial equivalence in the absense of logical operations dealing with restricted names.

Although the composition adjunct operation is certainly important for general context/system specifications, our work shows that the automated verification of concurrent systems using spatial logics that make essential use of the composition adjunct seems to be unfeasible. An important issue is then whether other expressive and tractable forms of contextual reasoning inspired by the composition adjunct, and extending those already provided by decidable behavioral-spatial logics, can be identified.

We thank Hongseok Yang for the illuminating discussion that prompted our counterexample in Section 4. We acknowledge Luís Monteiro, Daniel Hirschkoff and Davide Sangiorgi for all the rich exchanges and encouragement; and Luca Cardelli for many related discussions. E. Jeandel provided some references about quantifier elimination. This collaboration was supported by FET IST 2001-33310 Profundis. E. Lozes was also funded by an "Eurodoc" grant from *Région Rhône Alpes*.

References

- 1. S. Basu, R. Pollack, and M.-F. Roy. On the combinatorial and algebraic complexity of quantifier elimination. In *IEEE Symposium on Foundations of Computer Science*, 1994.
- L. Caires. Behavioral and Spatial Properties in a Logic for the Pi-Calculus. In Igor Walukiwicz, editor, *Proc. of Foundations of Software Science and Computation Structures*'2004, number 2987 in Lecture Notes in Computer Science. Springer Verlag, 2004.
- L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part II). In CONCUR 2002 (13th International Conference), number 2421 in Lecture Notes in Computer Science. Springer-Verlag, 2002.
- 4. L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part I). *Information and Computation*, 186(2):194–235, 2003.
- 5. L. Caires and E. Lozes. Elimination of Quantifiers and Undecidability in Spatial Logics for Concurrency. Technical report, ENS-Lyon LIP Report, 2004.
- 6. C. Calcagno, L. Cardelli, and A. D. Gordon. Deciding Validity in a Spatial Logic of Trees. In *ACM Workshop on Types in Language Design and Implementation*, pages 62–73, New Orleans, USA, 2003. ACM Press.
- C. Calcagno, H. Yang, and O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In Hariharan, Mukund, and Vinay, editors, *Proc. of FST TCS'2001*, volume 2245 of *Lecture Notes in Computer Sci*ence. Springer-Verlag, 2001.
- L. Cardelli, P. Gardner, and G. Ghelli. Manipulating Trees with Hidden Labels. In A. D. Gordon, editor, *Proceedings of the Sixth International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '03)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

- L. Cardelli and G. Ghelli. A Query Language Based on the Ambient Logic. In D. Sands, editor, 10th European Symposium on Programming (ESOP 2001), volume 2028 of Lecture Notes in Computer Science, pages 1–22. Springer-Verlag, 2001.
- L. Cardelli and A. Gordon. Logical Properties of Name Restriction. In S. Abramsky, editor, *Typed Lambda Calculi and Applications*, number 2044 in Lecture Notes in Computer Science. Springer-Verlag, 2001.
- 11. L. Cardelli and A. D. Gordon. Anytime, Anywhere. Modal Logics for Mobile Ambients. In 27th ACM Symp. on Principles of Programming Languages, pages 365–377. ACM, 2000.
- 12. W. Charatonik, A. D. Gordon, and J.-M. Talbot. Finite-control mobile ambients. In D. Metayer, *editor*, *11th European Symposium on Programming (ESOP 2002)*, number 2305 in Lecture Notes in Computer Science. Springer-Verlag, 2002.
- W. Charatonik and J.-M. Talbot. The decidability of model checking mobile ambients. In *Proceedings of the 15th Annual Conference of the European Association for Computer Science Logic*, Lecture Notes in Computer Science. Springer-Verlag, 2001.
- G. Conforti and G. Ghelli. Decidability of Freshness, Undecidability of Revelation. In Igor Walukiwicz, editor, *Proc. of Foundations of Software Science and Computation Structures*'2004, number 2987 in Lecture Notes in Computer Science. Springer Verlag, 2004.
- D. Hirschkoff. An Extensional Spatial Logic for Mobile Processes. In CON-CUR 2004 (15th International Conference), Lecture Notes in Computer Science. Springer-Verlag, 2004.
- D. Hirschkoff, E. Lozes, and D. Sangiorgi. Separability, Expressiveness and Decidability in the Ambient Logic. In *Third Annual Symposium on Logic in Computer Science*, Copenhagen, Denmark, 2002. IEEE Computer Society.
- D. Hirschkoff, E. Lozes, and D. Sangiorgi. Minimality results for the spatial logics. In *Proc. FSTTCS'2003*, number 2914 in Lecture Notes in Computer Science. Springer Verlag, 2003.
- E. Lozes. Adjunct elimination in the static Ambient Logic. In Proc. of EX-PRESS'2003, 2003. to appear in ENTCS, Elsevier.
- P. O'Hearn. Resources, Concurrency, and Local Reasoning (Abstract). In D. Schmidt, editor, *Proc. of ESOP*'2004, Lecture Notes in Computer Science, pages 1–2. Springer, 2004.
- J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In Seventieth Annual Symposium on Logic in Computer Science, Copenhagen, Denmark, 2002. IEEE Computer Society.
- D. Sangiorgi. Extensionality and Intensionality of the Ambient Logics. In 28th Annual Symposium on Principles of Programming Languages, pages 4–13. ACM, 2001.
- 22. B.A. Trakhtenbrot. The impossibility of an algorithm for the decision problem for finite models. *Dokłady Akademii Nauk SSR*, pages 70:569–572, 1950.

Modular Construction of Modal Logics

Corina Cîrstea¹ and Dirk Pattinson²

¹ School of Electronics and Computer Science, University of Southampton, UK cc2@ecs.soton.ac.uk

² Institut für Informatik, LMU München, Germany pattinso@informatik.uni-muenchen.de

Abstract. We present a modular approach to defining logics for a wide variety of state-based systems. We use coalgebras to model the behaviour of systems, and modal logics to specify behavioural properties of systems. We show that the syntax, semantics and proof systems associated to such logics can all be derived in a modular way. Moreover, we show that the logics thus obtained inherit soundness, completeness and expressiveness properties from their building blocks. We apply these techniques to derive sound, complete and expressive logics for a wide variety of probabilistic systems.

1 Introduction

Modularity has been a key concern in software engineering since the conception of the discipline [21]. This paper investigates modularity not in the context of building software systems, but in connection with specifying and reasoning about systems. Our work focuses on reactive systems, which are modelled as coalgebras over the category of sets and functions. The coalgebraic approach provides a uniform framework for modelling a wide range of state-based and reactive systems [27]. Furthermore, coalgebras provide models for a large class of probabilistic systems, as shown by the recent survey [3], which discusses the coalgebraic modelling of eight different types of probabilistic systems.

In the coalgebraic approach, a system consists of a state space C and a function $\gamma: C \to TC$, which maps every state $c \in C$ to the observations $\gamma(c)$ which can be made of c after one transition step. Different types of systems can then be represented in the by varying the type T of observations. A closer look at the coalgebraic modelling of state based and reactive systems reveals that in nearly all cases of interest, the type T of observations arises as the composition of a small number of basic constructs.

The main goal of this paper is to lift this compositionality at the level of observations to the level of specification languages and proof systems. That is, we associate a specification language and a proof system to every basic construct and show, how to obtain specification languages and proof systems for a combination of constructs in terms of the ingredients of the construction. Our main technical contribution is the study of the properties, which are preserved by a combination of languages and proof systems. On the side of languages, we isolate

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 258-275, 2004.

a property which ensures that combined languages are expressive, i.e. have the Hennessy-Milner property w.r.t. behavioural equivalence. Since this property is present in all of the basic constructs, we automatically obtain expressive specification languages for a large class of systems. Concerning proof systems, our main interests are soundness and completeness of the resulting logical system. In order to guarantee both, we investigate conditions which ensure that soundness and completeness of a combination of logics is inherited from the corresponding properties of the ingredients of the construction. Again, we demonstrate that this property is present in all basic building blocks.

As an immediate application of our compositional approach, we obtain sound, complete and expressive specification logics for a large class of probabilistic systems. To the best of the authors' knowledge, this class contains many systems, for which neither a sound and complete axiomatisation nor the Hennessy-Milner property was previously established, e.g. the simple and general probabilistic automata of Segala [28].

Our main technical tool to establish the above results is the systematic exploitation of the fact that coalgebras model the one-step behaviour of a system, i.e. that one application of the coalgebra map allows us to extract information about one transition step. This one-step behaviour of systems is parallelled both on the level of specification languages and proof systems. Regarding specification languages, we introduce the notion of *syntax constructor*, which specifies a set of syntactic features allowing the formulation of assertions about the next transition step of a system. Similarly, a *proof system constructor* specifies how one can infer judgements about the next transition step.

These notions are then used to make assertions about the global system behaviour by viewing the behaviour as the stratification of the observations which can be made after a (finite) number of steps. This is again parallelled on the level of the languages and proof systems. Completeness, for example, can then be established by isolating the corresponding one-step notion, which we call *one-step completeness*, and then proving that this entails completeness in the ordinary sense by induction on the number of transition steps. Expressiveness and soundness are treated similarly by considering the associated notions of one-step expressiveness and one-step soundness. When combining the logics, we combine both the syntax constructors and the proof system constructors, and show, that such combinations preserve one-step soundness, completeness and expressiveness.

The combination of logics and specification languages has been previously studied in different contexts. In the area of algebraic specification [30], structured specifications are used to combine already existing specifications along with their proof systems, see [4,6]. The main technique is the use of colimits in a category of algebraic signatures and corresponding constructions on the level of models and proof systems. Since the coalgebraic approach uses endofunctors to describe the behaviour of systems, our notion of signature is much richer, and we can accordingly investigate more constructions, with functor composition being the prime example. Furthermore, the coupling of the language and its semantics

is much stronger in the algebraic approach, due to the particular notions of signature and model (there is a 1-1 correspondence between function symbols on the syntactical side and functions on the level of models), so the (dual) notion of expressiveness does not play a role there.

The combination of logical systems has also been studied in its own right, based on Gabbay's notion of fibring logics [11]. The result of fibring two logics is a logic, which freely combines the connectives and proof rules from both logics. One is interested in the preservation of soundness and, in particular, completeness [32,7]. Our approach differs from fibring in that we consider a set of particular combinations of logical operators. These combinations are also of a very specific nature, since they allow to specify information about one transition step of the system. This makes our approach specific to coalgebras and modal logics, and allows us to use induction on the number of transition steps as a proof tool.

Finally, modal logics for coalgebras have been investigated by a number of authors, starting with Moss [20], who describes an abstract syntax for a large class of systems, but there is no general completeness result. Concrete logics for coalgebras and complete proof systems are described in [20,16,26,13]. This approach applies to an inductively defined class of systems, which is strictly subsumed by our approach, since we also obtain logics for probabilistic systems. Furthermore, thanks to the modularity of our construction, our logics are easily extensible to accommodate more features of transition systems, whereas it is *a priori* difficult to extend the approach of *loc. cit.* as one would have to work through one large inductive proof.

Regarding further work, we plan to extend our approach to more expressive logics, in particular to a coalgebraic version of CTL [9] and the modal μ calculus [15]. Also, it remains to be explored in what way our setup induces logics for programming languages with coalgebraically defined semantics [29,14,2].

2 Preliminaries and Notation

We denote the category of sets and functions by Set and pick a final object $1 = \{*\}$. Binary products (coproducts) in Set are written $X_1 \times X_2$ ($X_1 + X_2$), with canonical projections $\pi_i : X_1 \times X_2 \to X_i$ (canonical injections $\iota_i : X_i \to X_1 + X_2$). Finally, X^Y denotes the set of functions $Y \to X$.

We write Σ_{BA} for the algebraic signature specifying the boolean operators ff, tt, \neg , \rightarrow , \lor , \land . For any set X, its power set $\mathcal{P}X$ carries the structure of a Σ_{BA} -algebra. Then, for a set L and a function $d: L \rightarrow \mathcal{P}X$, we write \overline{L} for the carrier of the free Σ_{BA} -algebra over L, and $\overline{d}: \overline{L} \rightarrow \mathcal{P}X$ for the induced Σ_{BA} -morphism.

A boolean preorder (L, \vdash) is a Σ_{BA} -algebra L together with a preorder $\vdash \subseteq L \times L$ which is closed under the axioms and rules of propositional logic. The category of boolean preorders and order-preserving maps is denoted by Preord_{BA} ; the objects of Preord_{BA} are boolean preorders (L, \vdash) , while arrows from (L, \vdash) to (L', \vdash') are given by order-preserving Σ_{BA} -morphisms from L to L'.

We use endofunctors $T : Set \rightarrow Set$ to specify particular system types, and we refer to T sometimes as *signature functor*. More exactly, T specifies how the

information which can be observed of the system states *in one step* is structured. Systems themselves are then modelled as *T*-coalgebras.

Definition 1 (Coalgebras, Morphisms). A *T*-coalgebra is a pair (C, γ) where *C* is a set (the carrier, or state space of the coalgebra) and $\gamma : C \to TC$ a function (the coalgebra map, or transition structure). A coalgebra morphism $f : (C, \gamma) \to (D, \delta)$ is a function $f : C \to D$ such that $Tf \circ \gamma = \delta \circ f$. The category of *T*-coalgebras is denoted by CoAlg(*T*).

For $(C, \gamma) \in CoAlg(T)$, the transition structure determines the observations $\gamma(c) \in TC$ which can be made from a state $c \in C$ in one transition step. Morphisms between coalgebras preserve this one-step behaviour. The next example shows, that coalgebras can be used to model a wide variety of state-based and probabilistic systems:

Example 1. We use \mathcal{P} to denote the covariant powerset functor and \mathcal{D} for the probability distribution functor, given by

 $\mathcal{D}X = \{\mu : X \to [0,1] \mid \mu(x) = 0 \text{ for all but finitely many } x \in X \text{ and } \sum_{x \in X} \mu(x) = 1\}.$

(i) For $TX = \mathcal{P}(A \times X) \cong \mathcal{P}(X)^A$, it is easy to see that *T*-coalgebras $\gamma : C \to \mathcal{P}(A \times C)$ are in 1-1 correspondence with labelled transition systems (C, R) where $R \subseteq C \times A \times C$ is defined by $(c, a, c') \in R \iff (a, c') \in \gamma(c)$. Similarly, every \mathcal{P} -coalgebra determines a Kripke frame and vice versa.

(ii) Coalgebras for $TX = (1 + DX)^A$ are A-labelled probabilistic transition systems (see [10] for details).

(iii) The simple probabilistic automata and general probabilistic automata of [28] can be modelled as coalgebras for $TX = \mathcal{P}(A \times \mathcal{D}X)$ and $TX = \mathcal{P}(\mathcal{D}(A \times X))$.

Note that the endofunctors in the above examples are combinations of a small number of simple functors (constant, identity, powerset and probability distribution functor) using products, coproducts, exponentiation with finite exponents, and composition. In the sequel, we don't treat exponentiation with finite exponents explicitly, as it can be expressed using finite products. A recent survey of systems used in probabilistic modelling [3] identified no less than eight probabilistic system types of interest, all of which can be written as such a combination. Our goal is to derive languages and proof systems for these systems, using similar combinations on the logical level.

Apart from making this kind of compositionality explicit, the coalgebraic approach also allows for a uniform definition of behavioural equivalence, which specialises to standard notions of equivalence in many important examples.

Definition 2 (Behavioural Equivalence). Given T-coalgebras (C, γ) and (D, δ) , two states $c \in C$ and $d \in D$ are called behaviourally-equivalent (written $c \simeq d$) if there exist T-coalgebra morphisms f and g such that f(c) = g(d).

Two states c and d are ω -behaviourally equivalent (denoted by $c \simeq_{\omega} d$), if $\gamma_n(c) = \delta_n(d)$ for all $n \in \omega$, where, for $(E, \epsilon) \in \mathsf{CoAlg}(T)$, $\epsilon_0 : E \to 1$ is the unique map and $\epsilon_{n+1} = T\epsilon_n \circ \epsilon$.

The notion of ω -behavioural equivalence only takes finitely observable behaviour into account and is strictly weaker than behavioural equivalence. It can be shown that for ω -accessible T, both notions coincide [17]. It is often possible to define finitary logics for which logical equivalence coincides with ω -behavioural equivalence. On the other hand, we can not in general hope to characterise behavioural equivalence by a logic with finitary syntax.

It can be shown that for weak pullback preserving endofunctors, the notion of behavioural equivalence coincides with coalgebraic bisimulation, introduced by Aczel and Mendler [1] and studied by Rutten [27]. All functors considered in the sequel are weak pullback preserving. In the examples, the situation is as follows:

Example 2. We consider some of the systems introduced in Example 1.

(i) For labelled transition systems, i.e. coalgebras for $TX = \mathcal{P}(X)^A$, behavioural equivalence coincides with Park-Milner bisimulation [22,19].

(ii) The notion of behavioural equivalence for coalgebras for $TX = (1 + \mathcal{D}X)^A$, that is, probabilistic transition systems, coincides with the notion of probabilistic bisimulation considered in [18]. (This is proved in [10].)

A more detailed analysis of probabilistic systems from a coalgebraic point of view can be found in [3].

3 Modular Construction of Modal Languages

In this section we introduce *syntax constructors* and the modal languages they define. If we consider a modal language \mathcal{L} as an extension of prepositional logic, the idea of a syntax constructor is that it describes what we need to add to the prepositional language in order to obtain \mathcal{L} . The important feature of syntax constructors is, that they can be combined like the signature functors which define the particular shape of the systems under consideration. After introducing the abstract concept, we give examples of syntax constructors for some basic functors and show how they can be combined in order to obtain more structured modal languages.

Definition 3 (Syntax Constructor and Induced Language).

(i) A syntax constructor is an ω -accessible endofunctor $S : Set \to Set$, which preserves inclusions, i.e. $SX \subseteq SY$ for all $X \subseteq Y$.

(ii) The language $\mathcal{L}(S)$ associated with a syntax constructor is the least set of formulas containing

- ff and $\varphi \to \psi$ whenever $\varphi, \psi \in \mathcal{L}(S)$ - all $\varphi \in S(\Phi)$ whenever $\Phi \subseteq \mathcal{L}(S)$.

The requirement that syntax constructors preserve inclusions is mainly for ease of exposition, since in this case they define a monotone operator on sets, and languages can be constructed as least fixed points in the usual way. Equivalently, one could drop the requirement of inclusion-preservation at the expense of having to work with abstract (first oder) syntax, that is, constructing the language associated with a syntax constructor as the initial algebra of the functor $LX = 1 + X^2 + SX$.

Recall that an inclusion preserving endofunctor is ω -accessible iff, for all sets X and all $x \in TX$, there is a finite $S \subseteq X$ with $x \in TS$. Hence the requirement of ω -accessibility ensures that the construction of the associated language terminates after ω steps, that is, we are dealing with finitary logics only.

Before we show how syntax constructors can be combined, we introduce syntax constructors for some simple languages.

Example 3. (i) If A is a set (of atomic propositions), then the constant functor $S_A X = A$ is a syntax constructor. The associated language $\mathcal{L}(S)$ is the set of propositional formulas over the set A of atoms.

(ii) If M is a (possibly infinite) set of modal operators with associated (finite) arities, then S_M is a syntax constructor, where S_M maps a set X (of formulas) to the set $S_M(X)$ of formal expressions, given by

 $\mathsf{S}_{\mathsf{M}}(X) = \{ m(x_1, \dots, x_n) \mid m \in \mathsf{M} \text{ is } n \text{-ary}, x_1, \dots, x_n \in X \}.$

Viewing M as an algebraic signature, $S_M(X)$ is the set of terms with exactly one function symbol applied to variables in X. In the literature on modal logic, M is also called a modal similarity type [5]. The language of S_M is the set of modal formulas with modalities in M over the empty set of variables. For later reference, we let $S_P = S_{\{\Box\}}$ where \Box has arity one, and $S_D = S_M$ where M = $\{L_p \mid p \in \mathbb{Q} \cap [0,1]\}$, each L_p having arity one, and \mathbb{Q} denotes the set of rational numbers. The language associated with S_P is standard modal logic over the empty set of propositional variables. The language associated with S_D has a countable number of unary modalities, and will be used to describe probabilistic transition systems.

We are now ready for the first modularity issue of the present paper: the combination of syntax constructors to build more powerful languages from simple ingredients.

Definition 4 (Combinations of Syntax Constructors). Consider the following operations on sets L_1, L_2 (of formulas):

$$L_1 \otimes L_2 = \{ \pi(\varphi_1, \varphi_2) \mid \varphi_i \in L_i, i = 1, 2 \} \qquad L_1 \oplus L_2 = \{ \langle \kappa_i \rangle \varphi_i \mid \varphi_i \in L_i, i = 1, 2 \}.$$

For syntax constructors S_1, S_2 we let

$$(\mathsf{S}_1\otimes\mathsf{S}_2)X=\overline{\mathsf{S}_1X}\otimes\overline{\mathsf{S}_2X}\quad(\mathsf{S}_1\oplus\mathsf{S}_2)X=\overline{\mathsf{S}_1X}\oplus\overline{\mathsf{S}_2X}\quad(\mathsf{S}_1\otimes\mathsf{S}_2)X=\mathsf{S}_1(\overline{\mathsf{S}_2X}).$$

Note that above operations are of a purely syntactical nature, and the addition of the symbols π and κ_i serves as a way to ensure that the resulting functors are inclusion-preserving.

When combining syntax constructors, we add another layer of modal operators to already defined syntax. Closure under propositional connectives is needed to express propositional judgements also at the level on which the construction operates, e.g. to have formulas $\pi(\Box \varphi \lor \Box \psi, \Box \rho)$ in $\mathcal{L}(S_{\mathcal{P}} \otimes S_{\mathcal{P}})$.

The above definition is modelled after the definition of signature functors. In contrast to the logics treated in [26,13], our syntax constructors do not deal with exponentiation. This is due to the fact that infinite exponents fail to be ω -accessible, whereas finite exponents can be simulated by finite products. The third clause dealing with the composition of syntax constructors gives rise to S₁-modal operators which are indexed by S₂-formulas. Alternatively, the composition of syntax constructors can be thought of as introducing an additional sort:

Example 4. Suppose $S_i X = \{\Box_i x \mid x \in X\}$ for i = 1, 2. Then the language $\mathcal{L} = \mathcal{L}(S_1 \otimes S_2)$ can be described by the following grammar:

$$\mathcal{L} \ni \varphi, \psi ::= \mathrm{ff} \mid \varphi \to \psi \mid \Box_1 \rho \qquad (\rho \in \mathcal{L}') \\ \mathcal{L}' \ni \rho, \sigma ::= \mathrm{ff} \mid \sigma \to \rho \mid \Box_2 \varphi \qquad (\varphi \in \mathcal{L})$$

Languages of this kind can be used to specify properties of systems, whose signature functor T is the composition of two functors $T = T_1 \circ T_2$. In order to capture all possible behaviour described by T, we first have to describe the T_2 behaviour, and then use these descriptions to specify the observations which can be made according to T_1 . Since propositional connectives will be in general necessary to capture all possible T_2 behaviour, the definition of the syntax constructor $S_1 \otimes S_2$ involves the closure under propositional connectives before applying S_1 .

Similarly, languages of form $\mathcal{L}(S_1 \otimes S_2)$ and $\mathcal{L}(S_1 \oplus S_2)$ will be used to formalise properties of systems whose signature functors are ofform $T_1 \times T_2$ and $T_1 + T_2$, respectively. The next proposition shows that the constructions in Definition 4 indeed give rise to syntax constructors:

Proposition 1. $S_1 \otimes S_2$, $S_1 \oplus S_2$, $S_1 \otimes S_2$ are syntax constructors.

In ordinary modal logic, the modal language \mathcal{L} can be viewed as stratification $\mathcal{L} = \bigcup_{n \in \omega} \mathcal{L}^n$, where \mathcal{L}^n contains all modal formulas of rank $\leq n$. This in particular allows us to use induction on the rank of formulas as a proof principle.

Definition 5. Suppose S is a syntax constructor. Let $\mathcal{L}^0(S) = \overline{\emptyset}$ and $\mathcal{L}^{n+1}(S) = \overline{S(\mathcal{L}^n)}$. If $\varphi \in \mathcal{L}^n(S)$, we say that φ has rank at most n.

If $S = S_M$ for a set M of modal operators, then $\mathcal{L}^n(S)$ contains the modal formulas, whose depth of modal operators is at most n. The fact that $\mathcal{L}(S)$ can be viewed as a stratification of $\mathcal{L}^n(S)$, for $n \in \omega$, is the content of the next lemma.

Lemma 1. $\mathcal{L}(S) = \bigcup_{n \in \omega} \mathcal{L}^n(S)$ and $\overline{S(\mathcal{L}(S))} = \mathcal{L}(S)$.

TEAM LING

4 Modular Construction of Coalgebraic Semantics

In the previous section, we have argued that a syntax constructor with associated language \mathcal{L} specifies those features which have to be added to propositional logic in order to obtain \mathcal{L} . In standard modal logic, this boils down to adding the operator \Box , which can be used to describe the observable behaviour after one transition step. Abstracting from this example, we now introduce the *one-step semantics* of a syntax constructor, which relates the additional modal structure (specified by a syntax constructor) to the observations (specified by a signature functor) which can be made of a system in one transition step.

Throughout the section, S denotes a syntax constructor and T is an endofunctor; recall that \overline{L} is the closure of the set L under propositional connectives. Wewrite $\overline{S} : \operatorname{Alg}(\Sigma_{BA}) \to \operatorname{Alg}(\Sigma_{BA})$ for the functor taking a Σ_{BA} -algebra L to the Σ_{BA} -algebra $\overline{S}(L)$, and a Σ_{BA} -morphism $t : L \to L'$ to the obvious extension of $S(t) : SL \to SL'$ to a Σ_{BA} -morphism. The following definition provides a semantics to syntax constructors. As we are dealing with extensions of propositional logic, we use algebras for the boolean signature as a notational vehicle.

Definition 6 (One-step Semantics). If L is a Σ_{BA} -algebra and X is a set, then an interpretation of L over X is a Σ_{BA} -morphism $d: L \to \mathcal{P}X$. A morphism between interpretations $d: L \to \mathcal{P}X$ and $d': L' \to \mathcal{P}X'$ is a pair (t, f) with $t: L \to L' a \Sigma_{BA}$ -morphism and $f: X' \to X$ afunction, such that $d' \circ t = f^{-1} \circ d$:



A one-step semantics $[S]^T$ of a syntax constructor S w.r.t. an endofunctor T maps interpretations of L over X to interpretations of $\overline{S}(L)$ over TX, in such a way that whenever $(t, f) : d \to d'$ is a morphism of interpretations, so is $(\overline{St}, Tf) : [S]^T(d) \to [S]^T(d')$. We omit the superscript on the one-step semantics if the associated endofunctor is clear from the context.

A one-step semantics provides the glue between a language constructor and an endofunctor. The requirement that $[S]^T$ preserves morphisms of interpretations ensures that $[S]^T$ is defined uniformly on interpretations. This will subsequently guarantee that the (yet to be defined) coalgebraic semantics of the induced language $\mathcal{L}(S)$ is adequate w.r.t. behavioural equivalence; that is, behaviourally-equivalent states of coalgebras cannot be distinguished using formulas of the language.

A variant of the notion of one-step semantics, which treats syntax and the associated interpretation in the same framework, was studied in [8]. For languages with unary modalities, a one-step semantics corresponds to a choice of predicate liftings [23,24].

The key feature of a one-step semantics of a syntax constructor is that it gives rise to a semantics of $\mathcal{L}(S)$ w.r.t. *T*-coalgebras, that is, it defines a satisfaction

relation between *T*-coalgebras and formulas of $\mathcal{L}(S)$. Furthermore, we can define a one-step semantics of a combination of syntax constructors in terms of the one-step semantics of the ingredients. Before describing these constructions, we provide one-step semantics for some simple syntax constructors.

Example 5. We define one-step semantics for the syntax constructors introduced in Example 3.

(i) Suppose A is a set. Then the function which maps an arbitrary interpretation to the unique interpretation extending the identity function on A is a one-step semantics of S_A w.r.t. the constant functor TX = A.

(ii) A one-step semantics for $S_{\mathcal{P}}$ w.r.t. \mathcal{P} is given by

 $\llbracket \mathsf{S}_{\mathcal{P}} \rrbracket(d) : \overline{\mathsf{S}_{\mathcal{P}}(L)} \to \mathcal{PPX} \qquad \llbracket \mathsf{S}_{\mathcal{P}} \rrbracket(d)(\Box \varphi) = \{ x \subseteq X \mid x \subseteq d(\varphi) \}.$

(iii) For the syntax constructor $S_{\mathcal{D}}$ associated with the probability distribution functor, we define a one-step semantics by

$$\llbracket S_{\mathcal{D}} \rrbracket(d) : \overline{S_{\mathcal{D}}(L)} \to \mathcal{PDX} \qquad \llbracket S_{\mathcal{D}} \rrbracket(d)(L_p \varphi) = \{ \mu : \sum_{x \in d(\varphi)} \mu(x) \ge p \}$$

where $d: L \to \mathcal{P}X$ in both cases.

We now return to the claim made at the beginning of this section and show, that a one-step semantics gives rise to an interpretation of the associated language $\mathcal{L}(S)$ over *T*-coalgebras.

Definition 7 (Coalgebraic Semantics). Suppose S is a syntax constructor with one-step semantics $[S]^T$, and $(C, \gamma) \in CoAlg(T)$.

The coalgebraic semantics $\llbracket \varphi \rrbracket = \llbracket \varphi \rrbracket_C \subseteq C$ of a formula $\varphi \in \mathcal{L}(S)$ w.r.t. a *T*-coalgebra (C, γ) is defined inductively on the structure of formulas by

$$\llbracket ff \rrbracket = \emptyset \qquad \llbracket \varphi \to \psi \rrbracket = (C \setminus \llbracket \varphi \rrbracket) \cup \llbracket \psi \rrbracket$$
$$\llbracket \sigma \rrbracket = \gamma^{-1} (\llbracket S \rrbracket^T (d_{\Phi})(\sigma)) \qquad (for \ \sigma \in S\Phi)$$

where we inductively assume that $\llbracket \varphi \rrbracket$ is already defined for $\varphi \in \Phi$, giving rise to the map $d_{\Phi} : \Phi \to \mathcal{P}(C), \varphi \mapsto \llbracket \varphi \rrbracket$. Given $c \in C$, we write $c \models_C \varphi$ for $c \in \llbracket \varphi \rrbracket_C$, and $\operatorname{Th}(c) = \{\varphi \in \mathcal{L}(S) \mid c \models_C \varphi\}$.

Before showing that this definition captures the standard interpretation of some known modal logics, we need to show that the coalgebraic semantics is well defined, as we can have $\sigma \in S\Phi$ and $\sigma \in S\Psi$ for two different Φ, Ψ .

Lemma 2. The coalgebraic semantics of $\mathcal{L}(S)$ is well defined, that is, for $(C, \gamma) \in CoAlg(T)$ and $\Phi, \Psi \subseteq \mathcal{L}(S)$, we have $[\![S]\!]^T(d_{\Phi})(\sigma) = [\![S]\!]^T(d_{\Psi})(\sigma)$ for all $\sigma \in S\Phi \cap S\Psi$.

Note that the definition of the coalgebraic semantics generalises the semantics of modal formulas, as well as the semantics of the formulas considered in [12]:

Example 6. (i) Consider the syntax constructor $S_{\mathcal{P}}$ defined in Example 3, and the associated semantics $[\![S_{\mathcal{P}}]\!]$ as in Example 5. The induced coalgebraic semantics w.r.t. (C, γ) is defined inductively by

$$c \models \Box \varphi \text{ iff } e \models \varphi \text{ for all } e \in \gamma(c)$$

This is the standard textbook semantics of modal logic [5].

(ii) Consider the syntax constructor $S_{\mathcal{D}}$ defined in Example 3, and the associated semantics $[S_{\mathcal{D}}]$ as in Example 5. The induced coalgebraic semantics w.r.t. (C, γ) is defined inductively by

$$c \models L_p \varphi \text{ iff } \gamma(c)(\llbracket \varphi \rrbracket) \ge p$$

The above example shows that the coalgebraic semantics specialises to known semantics in concrete cases. We now turn to the issue of combining semantics, and show that we can derive a one-step semantics for a combination of syntax constructors (see Definition 4) by combining one-step semantics for the ingredients.

Definition 8 (Combinations of One-step Semantics). Let $d_1 : L_1 \to \mathcal{P}X_1$ and $d_2 : L_2 \to \mathcal{P}X_2$ be interpretations of L_1 (respectively L_2) over X_1 (respectively X_2) and consider the functions

$$\begin{aligned} &d_1 \otimes d_2 : L_1 \otimes L_2 \to \mathcal{P}(X_1 \times X_2), \ \pi(\varphi_1, \varphi_2) \mapsto \{(x_1, x_2) \mid x_i \in d_i(\varphi_i)\} \\ &d_1 \oplus d_2 : L_1 \oplus L_2 \to \mathcal{P}(X_1 + X_2), \ \langle \kappa_i \rangle \varphi_i \mapsto \{\iota_i(x_i) \mid x_i \in d_i(\varphi_i)\}. \end{aligned}$$

If $[S_i]^{T_i}$ is a one-step semantics of a syntax constructor S_i w.r.t. an endofunctor T_i , for i = 1, 2, the one-step semantics of various combinations of S_1 and S_2 is given as follows:

$$\llbracket \mathsf{S}_1 \otimes \mathsf{S}_2 \rrbracket(d) = \llbracket \mathsf{S}_1 \rrbracket(\overline{d}) \otimes \llbracket \mathsf{S}_2 \rrbracket(\overline{d}) \qquad \llbracket \mathsf{S}_1 \oplus \mathsf{S}_2 \rrbracket(d) = \llbracket \mathsf{S}_1 \rrbracket(d) \oplus \llbracket \mathsf{S}_2 \rrbracket(d)$$
$$\llbracket \mathsf{S}_1 \otimes \mathsf{S}_2 \rrbracket(d) = \llbracket \mathsf{S}_1 \rrbracket(\llbracket \mathsf{S}_2 \rrbracket(d))$$

where we have notationally suppressed that $[S_1] \circ [S_2]$ is a one-step semantics of $S_1 \circ S_2$ w.r.t. $T_1 \circ p' T_2$, for $\circ p = \otimes, \oplus, \otimes$ and $\circ p' = \times, +, \circ$.

Note the absence of the closure operator $\overline{\cdot}$ in the last clause; this is already taken care of by the definition of $S_1 \otimes S_2$. The intuitions behind the definitions of $d_1 \oplus d_2$, $d_1 \otimes d_2$ are as follows. Assuming that L_1 and L_2 are interpreted over X_1 and X_2 , respectively, we can interpret the language $L_1 \otimes L_2$ (respectively $L_1 \oplus L_2$) over $X_1 \times X_2$ (respectively $X_1 + X_2$). In the first case, a formula $\pi(\varphi_1, \varphi_2)$ holds at a state $x = (x_1, x_2)$ iff φ_i holds in x_i , i = 1, 2. Also, $\langle \kappa_i \rangle \varphi_i$ holds in $x \in X_1 + X_2$ iff $x = \iota_i(x_i)$ and φ_i holds in x_i .

We now show that the combination of one-step semantics is well defined. To make notation bearable we disregard the dependency on the endofunctor.

Proposition 2. Suppose $[S_i]$ is a one-step semantics for S_i w.r.t. T_i for i = 1, 2. Then $[S_1] \otimes [S_2]$, $[S_1] \oplus [S_2]$ and $[S_1] \otimes [S_2]$ are one-step semantics for $T_1 \times T_2$, $T_1 + T_2$ and $T_1 \circ T_2$, respectively. We have therefore seen how we can combine syntax constructors and their associated one-step semantics. This gives rise to a modular way of constructing languages for coalgebras. The following two sections present applications of the modular approach. In the next section we show that a combination of logics has the Hennessy-Milner property if all the ingredients satisfy an expressiveness property. In the subsequent section, we show how to obtain sound and complete proof systems for a combination of logics by suitably combining sound and complete proof systems for the building blocks.

5 Behavioural Versus Logical Equivalence

In this section, we show that any two behaviourally equivalent points necessarily have the same logical theory. In order to prove the Hennessy-Milner property for a logic which arises from a combination of syntax constructors, we introduce the notion of expressiveness for an interpretation $L \rightarrow \mathcal{P}X$, and show that the language associated with a one-step semantics which preserves expressiveness has the Hennessy-Milner property. To treat languages which arise from a combination of syntax constructors, we show that the combination of one-step semantics preserves expressiveness if all of the ingredients do. This in particular allows us to establish the Hennessy-Milner property for combined languages in a modular fashion. We begin with the easy part and show that behaviourally equivalent states cannot be distinguished by formulas of a logic which is induced by a syntax constructor.

Proposition 3. Suppose S is a syntax constructor with one-step semantics [S], and $(C, \gamma), (D, \delta) \in \mathsf{CoAlg}(T)$. Then, $\mathrm{Th}(c) = \mathrm{Th}(d)$ whenever $c \simeq d$.

The remainder of the section is concerned with the converse of Proposition 3. For that, we introduce the notion of one-step expressiveness, which allows to derive a Hennessy-Milner property for the language associated with a syntax constructor. Moreover, we show that this condition automatically holds for a combination of syntax constructors, if it is valid for the ingredients of the construction.

Definition 9 (One-Step Expressiveness).

(i) An interpretation $d : L \to \mathcal{P}X$ is expressive if the associated language map $d^{\sharp} : X \to \mathcal{P}L$, given by $x \mapsto \{\varphi \in \mathcal{L} : x \in d(\varphi)\}$ is injective. (ii) A one-step semantics $[S]^T$ is one-step expressive if $[S]^T(d)$ is expressive

(ii) A one-step semantics $[S]^T$ is one-step expressive if $[S]^T(d)$ is expressive whenever d is.

Using this terminology, our first main result can be stated as follows:

Theorem 1. If $[S]^T$ is one-step expressive, then $\mathcal{L}(S)$ is expressive w.r.t. \simeq_{ω} , i. e. $\operatorname{Th}(c) = \operatorname{Th}(d)$ iff $c \simeq_{\omega} d$ for all $(C, \gamma), (D, \delta) \in \operatorname{CoAlg}(T)$ and $(c, d) \in C \times D$.

In other words, the logic is strong enough to distinguish all states, which exhibit different behaviour, which can be witnessed by observing finitely many steps only. The proof of this theorem uses induction on the rank of formulas (see Definition 5), and a semantical representation of a formula of rank n as a subset of T^n 1. Using the fact that ω -behavioural equivalence coincides with behavioural equivalence for coalgebras of an ω -accessible endofunctor (see [31]), we have the following corollary:

Corollary 1. If T is ω -accessible, then $\mathcal{L}(S)$ is expressive, that is, $\operatorname{Th}(c) = \operatorname{Th}(d)$ iff $c \simeq d$ for all (C, γ) , $(D, \delta) \in \operatorname{CoAlg}(T)$ and all $(c, d) \in C \times D$.

Note that the accessibility degree of the underlying endofunctor T basically limits the branching degree of T-coalgebras [24], so the above corollary is a coalgebraic Hennessy-Milner result.

It is easy to see that the one-step semantics of all basic syntax constructors are one-step expressive:

Example 7. The one-step semantics of the syntax constructors from Example 5 are one-step expressive, if we consider the *finite* powerset functor \mathcal{P}_f in clause (ii).

Our next goal is to show that one-step expressiveness is preserved by all the combinations of syntax constructors. Again suppressing the dependency on the endofunctor T we obtain:

Proposition 4. Suppose $[S_i]$ are one-step expressive, for i = 1, 2. Then so are $[S_1] \otimes [S_2]$, $[S_1] \oplus [S_2]$ and $[S_1] \otimes [S_2]$.

Thus, Theorem 1 applies to any combination of one-step semantics which are one-step expressive. Note that this in particular implies that the language associated with the combination of two syntax constructors distinguishes any two states up to ω -behavioural equivalence, or in case *T* is ω -accessible, even up to behavioural equivalence. As an immediate application, we obtain expressive languages for all system types discussed in Example 1.

6 Modular Construction of Proof Systems

This section extends the methods presented so far to also include the compositional construction of proof systems. Our main result shows that this can be done in such a way that the combined proof system inherits soundness and completeness from its building blocks. The key notion needed to formulate the modularisation of proof systems is that of a proof system constructor.

Definition 10 (Proof System Constructor). Suppose S is a syntax constructor. A proof system constructor for S is a functor $P : Preord_{BA} \rightarrow Preord_{BA}$ such that

(i) $\overline{\mathsf{S}} \circ U = U \circ \mathsf{P}$, where $U : \mathsf{Preord}_{\mathsf{BA}} \to \mathsf{Alg}(\Sigma_{\mathsf{BA}})$ is the forgetful functor;

(ii) P preserves order-reflecting morphisms.

270 C. Cîrstea and D. Pattinson

The intuition is as follows. The syntax constructor S specifies a set of modalities to be added to propositional logic, while the induced functor \overline{S} produces the language which arises by applying the given modal operators exactly once, and subsequently closing under propositional connectives. Now a corresponding proof system constructor takes a boolean preorder $\vdash_L \subseteq L \times L$, which represents all facts that can be proved about formulas in L, and produces a boolean preorder $\vdash_{\overline{S}(L)} \subseteq \overline{S}(L) \times \overline{S}(L)$, which defines all provable sequents over the next transition step, that can be derived from sequents over L. In other words, a proof system constructor specifies how we can lift sequents to formulas containing an extra degree of nesting of the modal operators. The second requirement in Definition 10 formalises a well-behavedness property of proof system constructors, which will ensure that the proof systems induced by proof system constructors can be constructed inductively.

Since the axioms of modal logic involve formulas of rank one only, we can give a straightforward encoding of modal logic in a proof system constructor.

Example 8. Consider the syntax constructor $S_{\mathcal{P}}$ defined in Example 3. For a boolean preorder (L, \vdash_L) , define $P_{\mathcal{P}}(L, \vdash_L) = (\overline{S_{\mathcal{P}}}(L), \vdash_{\overline{S_{\mathcal{P}}}(L)})$ where $\vdash_{\overline{S_{\mathcal{P}}}(L)}$ is the relation generated by the following axioms and rules:

$$\frac{\varphi \vdash_L \psi}{\Box \varphi \vdash_{\overline{S_{\mathcal{P}}}(L)} \Box \psi} \quad \text{tt} \vdash_{\overline{S_{\mathcal{P}}}(L)} \Box \text{tt} \quad \Box \varphi \land \Box \psi \vdash_{\overline{S_{\mathcal{P}}}(L)} \Box (\varphi \land \psi)$$

augmented with the axioms and rules of propositional logic. Then $P_{\mathcal{P}}$ is a proof system constructor for $S_{\mathcal{P}}$.

In the case of probabilistic transition systems, the logic in [12] can also be captured by a proof system constructor.

Example 9. Consider the syntax constructor $S_{\mathcal{D}}$ defined in Example 3. For $p \in [0,1]$, let $M_p \varphi ::= L_{1-p} \neg \varphi$, and $E_p \varphi ::= L_p \varphi \land M_p \varphi$. Also, for a finite sequence of formulas $\varphi_1, \ldots, \varphi_m$, let $\varphi^{(k)}$ stand either for $\bigvee_{\substack{1 \leq l_1 < \ldots < l_k \leq m}} (\varphi_{l_1} \land \ldots \land \varphi_{l_k})$, if

 $k \leq m$, or for ff, if k > m. Thus, the formula $\varphi^{(k)}$ states that, from among the formulas $\varphi_1, \ldots, \varphi_m$, at least k are true at any point.

Now for each boolean preorder (L, \vdash) , define $\mathsf{P}_{\mathcal{D}}(L, \vdash) = (\overline{\mathsf{S}_{\mathcal{D}}}(L), \vdash')$, where the relation \vdash' is generated by the axioms and rules in Figure 1, augmented with the axioms and rules of propositional logic. All but the last of these axioms

$$\begin{array}{cccc} \vdash' L_0 \varphi & \vdash' L_p \mathsf{tt} & \vdash' \neg L_p \varphi \to M_p \varphi & \vdash' L_p \varphi \to \neg L_q \neg \varphi \\ \\ \frac{\varphi \vdash \psi}{L_p \varphi \vdash' L_p \psi} & \frac{\vdash \bigwedge_{k=1}^{max(m,n)} \varphi^{(k)} \leftrightarrow \psi^{(k)}}{\vdash' (\bigwedge_{i=1}^m L_{p_i} \varphi_i) \land (\bigwedge_{j=2}^n M_{q_i} \psi_i) \to L_{p_1 + \ldots + p_m - (q_2 + \ldots + q_n)} \psi_1 \end{array}$$

Fig. 1. Axioms and Rules for \mathcal{D} , where p + q > 1

and rules capture immediate properties of the one-step semantics $[S_{\mathcal{D}}]$ defined in Example 5. The last rule describes a more involved property of probability distributions, see [12] for details.

The functor $P_{\mathcal{D}}$: Preord_{BA} \rightarrow Preord_{BA} defined above qualifies as a proof system constructor for $S_{\mathcal{D}}$.

A proof system constructor P for S induces a derivability relation \vdash_{P}^{g} on the language $\mathcal{L}(S)$, defined as the set of judgements, which one can infer by applying the proof system constructor.

Definition 11 (Global Proof System Induced by P). *The* global proof system induced by P *is the least boolean preorder* $(\mathcal{L}(S), \vdash_{P}^{g})$ *such that* $P(\mathcal{L}(S), \vdash_{P}^{g}) \subseteq (\mathcal{L}(S), \vdash_{P}^{g})$.

In particular, since \vdash_{p}^{g} is a boolean preorder, it contains all instances of propositional tautologies. We now apply our main programme also to this definition, and show, that the global proof system can be viewed as stratification of a sequence of relations $\vdash_{p}^{p} \subseteq \mathcal{L}^{n}(S) \times \mathcal{L}^{n}(S)$. This will open the road for the proof of soundness and completeness using induction on the rank of the formulas.

Definition 12 (Inductive Proof System Induced by P). *For* $n \in \omega$, *define* $\vdash_n \subseteq \mathcal{L}^n(S) \times \mathcal{L}^n(S)$ *by:*

 $-\varphi \vdash_0 \psi \text{ whenever } \varphi \rightarrow \psi \text{ is a propositional tautology;} \\ -(\mathcal{L}^{n+1}(\mathsf{S}), \vdash_{n+1}) = \mathsf{P}(\mathcal{L}^n(\mathsf{S}), \vdash_n)$

The inductive proof system induced by P is given by $(\mathcal{L}(\mathsf{S}), \vdash_{\mathsf{P}}) = \bigcup_{n \in \omega} (\iota_n \times \iota_n)(\mathcal{L}^n(\mathsf{S}), \vdash_n)$ where $\iota_n : \mathcal{L}^n(\mathsf{S}) \to \mathcal{L}(\mathsf{S})$ denote the inclusions arising from Lemma 1.

Lemma 3. $(\mathcal{L}^n(S), \vdash_n) \subseteq (\mathcal{L}^{n+1}(S), \vdash_{n+1}) \text{ and } \mathsf{P}(\mathcal{L}(S), \vdash_{\mathsf{P}}) = (\mathcal{L}(S), \vdash_{\mathsf{P}}).$

The two requirements in the definition of proof system constructors are exactly what is needed to show that the two proof systems induced by P coincide.

Proposition 5. The boolean preorders $(\mathcal{L}(S), \vdash_{P}^{g})$ and $(\mathcal{L}(S), \vdash_{P})$ coincide.

We can therefore use induction on n to prove properties of $(\mathcal{L}(S), \vdash_{P}^{g})$. In the following, we consider soundness and completeness of $(\mathcal{L}(S), \vdash_{P}^{g})$ w.r.t. the coalgebraic semantics induced by some one-step semantics $[S]^{T}$, and show that these follow from soundness and completeness conditions involving $[S]^{T}$ and P.

Definition 13 (One-Step Soundness and Completeness).

(i) A boolean preorder $\vdash \subseteq L \times L$ is sound (complete) w.r.t. an interpretation $d: L \to \mathcal{P}X$ if $\varphi \vdash \psi$ implies $d(\varphi) \subseteq d(\psi)$ ($d(\varphi) \subseteq d(\psi)$ implies $\varphi \vdash \psi$) for any $\varphi, \psi \in L$.

(ii) A proof system constructor P for S is one-step sound (complete) w.r.t. a one-step semantics $[\![\mathsf{S}]\!]^T$ if $\mathsf{P}(L,\vdash)$ is sound (complete) w.r.t. $[\![\mathsf{S}]\!]^T(d) : \overline{\mathsf{S}}L \to \mathcal{P}TX$ whenever \vdash is sound (complete) w.r.t. $d : L \to \mathcal{P}X$. Using induction, we can derive soundness and completeness in the standard way from their one-step counterparts:

Theorem 2 (Soundness and Completeness). Assume $T : \text{Set} \to \text{Set}$ is such that $T1 \neq \emptyset$. If the proof system constructor P for S is one-step sound (complete) w.r.t. $[\mathsf{S}]^T$, then $(\mathcal{L}(\mathsf{S}), \vdash_{\mathsf{P}})$ is sound (complete) w.r.t. the coalgebraic semantics of $\mathcal{L}(\mathsf{S})$.

In the case of probabilistic transition systems, the axioms and rules given in Example 9 form a sound and complete proof system. This was proved in [12] using the standard filtration method. For us, this result is of limited usefulness, as we must show that the proof system constructor defined in Example 9 is one-step sound and complete. This will later allow us to derive sound and complete proof systems for more complex types of probabilistic systems.

The following proposition, which deals with the base case of the probability distribution functor, puts us into the position to apply our techniques to a large class of probabilistic systems.

Proposition 6. The proof system constructor $P_{\mathcal{D}}$ of Example 9 is one-step sound and complete w.r.t. $[S_{\mathcal{D}}]$.

The proof of this result makes use of Rockafellar's Theorem [25].

In what follows, we will show how one can combine proof system constructors for simple languages in order to derive proof systems for more complex languages. Moreover, we will show that whenever the building blocks of such constructions are one-step sound and complete w.r.t. some given one-step semantics, the resulting proof system is sound and complete w.r.t. the induced coalgebraic semantics. To describe the combinations of proof systems, we introduce the following notation:

$$\begin{split} & [\pi_1]\varphi_1 ::= \pi(\varphi_1, \mathsf{tt}) \in (\mathsf{S}_1 \otimes \mathsf{S}_2)X \\ & [\kappa_i]\varphi_i ::= \neg \langle \kappa_i \rangle (\neg \varphi_i) \in \overline{(\mathsf{S}_1 \oplus \mathsf{S}_2)X} \end{split} \qquad \begin{aligned} & [\pi_2]\varphi_2 ::= \pi(\mathsf{tt}, \varphi_2) \in (\mathsf{S}_1 \otimes \mathsf{S}_2)X \\ & \text{if } \varphi_i \in \overline{\mathsf{S}_iX} \text{ for } i = 1, 2. \end{aligned}$$

Definition 14 (Combinations of Proof System Constructors). Let (L_1, \vdash_1) and (L_2, \vdash_2) be boolean preorders.

(i) We let $(L_1, \vdash_1) \otimes (L_2, \vdash_2) = (\overline{L_1 \otimes L_2}, \vdash_{\otimes})$, where $L_1 \otimes L_2$ is as in Definition 4, and the relation \vdash_{\otimes} is generated by the following axioms and rules:

$$\vdash_{\otimes} [\pi_i] \mathsf{tt} \qquad [\pi_i] \varphi \wedge [\pi_i] \psi \vdash_{\otimes} [\pi_i] (\varphi \wedge \psi) \qquad \frac{\varphi \vdash_i \psi}{[\pi_i] \varphi \vdash_{\otimes} [\pi_i] \psi}$$

augmented with the axioms and rules of propositional logic.

(ii) We let $(L_1, \vdash_1) \oplus (L_2, \vdash_2) = (\overline{L_1 \oplus L_2}, \vdash_{\oplus})$, where $L_1 \oplus L_2$ is as in Definition 4, and the relation \vdash_{\oplus} is generated by the following axioms and rules:

$$\vdash_{\oplus} [\kappa_{i}] \mathsf{tt} \qquad [\kappa_{i}] \varphi \wedge [\kappa_{i}] \psi \vdash_{\oplus} [\kappa_{i}] (\varphi \wedge \psi)$$
$$[\kappa_{1}] \mathsf{ff} \wedge [\kappa_{2}] \mathsf{ff} \vdash_{\oplus} \mathsf{ff} \qquad \frac{\varphi \vdash_{i} \psi}{[\kappa_{i}] \varphi \vdash_{\oplus} [\kappa_{i}] \psi}$$

augmented with the axioms and rules of propositional logic.

If
$$\mathsf{P}_1$$
 and P_2 are proof system constructors for S_1 and S_2 , respectively, define:
 $(\mathsf{P}_1 \otimes \mathsf{P}_2)(L, \vdash) = \mathsf{P}_1(L, \vdash) \otimes \mathsf{P}_2(L, \vdash) \quad (\mathsf{P}_1 \oplus \mathsf{P}_2)(L, \vdash) = \mathsf{P}_1(L, \vdash) \oplus \mathsf{P}_2(L, \vdash)$
 $(\mathsf{P}_1 \otimes \mathsf{P}_2)(L, \vdash) = \mathsf{P}_1(\mathsf{P}_2(L, \vdash))$

With these definitions we obtain that soundness and completeness is preserved by combinations of proof system constructors; for readability we have suppressed the dependency of the one-step semantics on the endofunctor.

Proposition 7. Suppose P_i is a proof system constructor for S_i , for i = 1, 2. Then, $P_1 \otimes P_2$, $P_1 \oplus P_2$ and $P_1 \otimes P_2$ are proof system constructors for $S_1 \otimes S_2$, $S_1 \oplus S_2$ and $S_1 \otimes S_2$, respectively. Moreover, if P_1 and P_2 are one-step sound (complete) w.r.t. $[S_1]$ and $[S_2]$, respectively, then $P_1 \otimes P_2$, $P_1 \oplus P_2$ and $P_1 \otimes P_2$ are one-step sound (complete) w.r.t. $[S_1] \otimes [S_2]$, $[S_1] \oplus [S_2]$ and $[S_1] \otimes [S_2]$, respectively.

Note that, if P_1 and P_2 are defined in terms of axioms and rules, all their combinations can be described in the same way.

As we have already argued in the beginning, a large class of probabilistic systems can be modelled as coalgebras of signature functors of the following form:

 $T ::= A \mid \mathcal{I}d \mid \mathcal{P}_f \mid \mathcal{D} \mid T_1 \times T_2 \mid T_1 + T_2 \mid T_1 \circ T_2.$

We can therefore use Propositions 1, 2 and 7 to derive, for any probabilistic system type of the above form, a logic which is sound, complete and expressive.

Example 10 (Probabilistic Automata). Simple probabilistic automata [28] are modelled coalgebraically using the functor $TX = \mathcal{P}_f(A \times \mathcal{D}X)$. The language $\mathcal{L} = \mathcal{L}(T)$ obtained by applying the modular techniques presented earlier can be described by the following grammar:

$\mathcal{L} i \varphi ::= ff \mid \varphi ightarrow \varphi' \mid \Box \psi$	$(\psi \in \mathcal{L}')$
$\mathcal{L}' i \psi ::= ff \mid \psi ightarrow \psi' \mid (a,\xi)$	$(\xi \in \mathcal{L}'')$
$\mathcal{L}'' i \xi ::= \mathrm{ff} \mid \xi \to \xi' \mid L_p \varphi$	$(arphi \in \mathcal{L})$

The coalgebraic semantics and the associated proof system are similarly given by a three layer construction.

References

- 1. P. Aczel and N. Mendler. A final coalgebra theorem. In D. H. Pitt et al, editor, *Category Theory and Computer Science*, volume 389 of *LNCS*. Springer, 1989.
- 2. F. Bartels. *On generalised coinduction and probabilistic specification formats.* PhD thesis, CWI, Amsterdam, 2004.
- 3. F. Bartels, A. Sokolova, and E. de Vink. A hierarchy of probabilistic system types. In H.P. Gumm, editor, *Proc. CMCS 2003*, volume 82 of *ENTCS*. Elsevier, 2003.
- M. Bidoit, M. V. Cengarle, and R. Hennicker. Proof systems for structured specifications and their refinements. In H. J. Kreowski, E. Astesiano, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, IFIP State-ofthe-Art Reports, pages 385–434. Springer, 1999.
- 5. P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Cambridge University Press, 2000.
- 6. T. Borzyszkowski. Logical systems for structured specifications. *Theoret. Comput. Sci.*, 286(2):197–245, 2002.
- 7. C. Caleiro. Combining Logics. PhD thesis, 2002.
- 8. C. Cîrstea. On expressivity and compositionality in logics for coalgebras. In H.P. Gumm, editor, *Proc. CMCS 2003*, volume 82 of *ENTCS*. Elsevier, 2003.
- E. Clarke and E. Emerson. Synthesis of synchronisation skeletons for branching temporal logics. In *Workshop on Logics of Programs*, volume 131 of LNCS. Springer, 1981.
- 10. E. de Vink and J. Rutten. Bisimulation for probabilistic transition systems: a coalgebraic approach. *Theoret. Comput. Sci.*, 221:271–293, 1999.
- 11. D. Gabbay. Fibring Logics. Oxford University Press, 1998.
- A. Heifetz and P. Mongin. Probability logic for type spaces. *Games and Economic Behaviour*, 35:31–53, 2001.
- B. Jacobs. Many-sorted coalgebraic modal logic: a model-theoretic study. *Theor. Inform. Appl.*, 35(1):31–59, 2001.
- M. Kick. Bialgebraic modelling of timed processes. In P. Widmayer, F. Ruiz, R. Bueno, M. Hennessy, S. Eidenbenz, and R. Conejo, editors, *Proc. ICALP 2002*, volume 2380 of *LNCS*. Springer, 2002.
- D. Kozen. Results on the propositional mu-calculus. *Theoret. Comput. Sci.*, 27:333– 354, 1983.
- A. Kurz. Specifying Coalgebras with Modal Logic. *Theoret. Comput. Sci.*, 260(1–2):119–138, 2001.
- A. Kurz and D. Pattinson. Definability, canonical models, and compactness for finitary coalgebraic modal logic. In L.S. Moss, editor, *Proc. CMCS 2002*, volume 65 of *ENTCS*. Elsevier, 2002.
- K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Inform. and Comput.*, 94:1–28, 1991.
- 19. R. Milner. *Communication and Concurrency*. International series in computer science. Prentice Hall, 1989.
- 20. L.S. Moss. Coalgebraic logic. Ann. Pure Appl. Logic, 96:277-317, 1999.
- P. Naur and B. Randell, editors. Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968. Scientific Affairs Division, NATO, 1969.
- 22. D. Park. Concurrency and automata on infinite sequences. In *Proceedings of the* 5th GI Conference, volume 104 of LNCS. Springer, 1981.
- 23. D. Pattinson. Coalgebraic modal logic: Soundness, completeness and decidability of local consequence. *Theoret. Comput. Sci.*, 309(1–3):177–193, 2003.
- 24. D. Pattinson. Expressive logics for coalgebras via terminal sequence induction. *Notre Dame J. Formal Logic*, 2004. To appear.
- 25. R.T. Rockafellar. Convex Analysis. Princeton University Press, 1970.
- 26. M. Rößiger. From Modal Logic to Terminal Coalgebras. *Theoret. Comput. Sci.*, 260:209–228, 2001.
- 27. J.J.M.M. Rutten. Universal coalgebra: A theory of systems. *Theoret. Comput. Sci.*, 249:3–80, 2000.

- 28. R. Segala. *Modelling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Massachusetts Institute of Technology, 1995.
- D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *Proc.* 12th LICS Conference, pages 280–291. IEEE, Computer Society Press, 1999.
- 30. M. Wirsing. Algebraic specification. In J. van Leuween, editor, *Handbook of Theoretical Computer Science*. Elsevier, 1990.
- 31. J. Worrell. On the Final Sequence of an Accessible Set Functor. *Theoret. Comput. Sci.*, 2003. To appear.
- A. Zanardo, A. Sernadas, and C. Sernadas. Fibring: Completeness preservation. J. Symbolic Logic, 66(1):414–439, 2001.

Verification by Network Decomposition*

Edmund Clarke¹, Muralidhar Talupur¹, Tayssir Touili¹, and Helmut Veith²

¹ Carnegie Mellon University
 ² Technische Universität München

Abstract. We describe a new method to verify networks of homogeneous processes which communicate by token passing. Given an arbitrary network graph and an indexed $LTL \setminus X$ property, we show how to decompose the network graph into multiple constant size networks, thereby reducing one model checking call on a large network to several calls on small networks. We thus obtain cut-offs for arbitrary classes of networks, adding to previous work by Emerson and Namjoshi on the ring topology. Our results on $LTL \setminus X$ are complemented by a negative result which precludes the existence of reductions for $CTL \setminus X$ on general networks.

1 Introduction

Despite the big success of model checking in hardware and software verification, the classical approach to model checking can handle only finite state systems. Consequently, applying model checking techniques to systems involving unlimited concurrency, unlimited memory, or unlimited domain sizes, is a major challenge. Researchers have sought to address these issues by different verification methods including, among others, abstraction, regular model checking, static analysis and theorem proving.

Many software and hardware systems however are described in terms of natural parameters, and for each concrete value of the parameters, the systems have finite state space. A system model involving such parameters is called a parameterized system. Verifying a property of a parameterized system amounts to verifying this property for all values of the parameters. Examples of parameterized systems include, mutual exclusion protocols, cache coherence protocols and multi-threaded systems.

^{*} This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grants no. CCR-9803774 and CCR-0121547, the Office of Naval Research (ONR) and the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485 and by the European Community Research Training Network GAMES. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of SRC, NSF, ONR, NRL, ARO, the U.S. Government or any other entity.

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 276-291, 2004.

In a seminal paper, Emerson and Namjoshi [17] consider systems composed of identical asynchronous processes which are arranged in a ring topology and communicate by passing a boolean token. For several classes of indexed $CTL^* \setminus X$ properties [9] they provide *cutoffs*, i.e., reductions to single systems of constant small size. Consequently, $CTL^* \setminus X$ properties over an *infinite class of networks* can be reduced to a *single model checking call*.

In this paper, we extend the results of Emerson and Namjoshi from rings to *arbitrary classes of networks*. There are two modifications, however: first, our results hold true only for LTLX, and second, we introduce a more refined notion of cut-offs. The first restriction is necessary: We show in Section 4 that with $CTL \setminus X$ it is *impossible* to obtain cut-offs for arbitrary networks.

The second modification actually provides an interesting new view on the notion of cut-offs: in order to verify the parametrized system, we are allowed to model check a *constant number* c of small systems whose network graphs have sizes bounded by a *constant* s. Then, the verification result for the parametrized system is a Boolean combination of the collected results for the small systems. We call such a reduction to a finite case distinction a (c, s)-bounded reduction.

Our main results can be summarized as follows:

- Verification by Network Decomposition: Verifying systems with fixed large network graphs G (e.g., concrete instantiations of a parametrized system) can be as challenging as verifying parameterized systems. Note that when |Q| is the state space of the individual processes, then the state space of the whole network can be as high as $|Q|^n$, where n is the number of nodes. We show that the verification of an indexed LTLX property φ for a system with network graph G can be achieved by an *efficiently computable* (c, s)-bounded reduction. For the important case of 2-indexed properties, it is sufficient to model check at most 36 networks of size 4.
- Offline Verification: In a scenario where φ is known in advance and the network *G* can change for different applications, we can first verify a constant number of small systems *offline*. Later, when we get to know the network graph *G*, the correctness of *G* with respect to specification φ can be verified *online* by simply evaluating a constant size Boolean function, regardless of the size of the processes.

Again, for 2-indexed properties, the offline computation involves at most 36 calls to the model checker for networks of size 4.

- **Cut-Offs:** For every class of networks \mathbb{T} and *k*-indexed LTL\X property φ one can verify if φ holds on *all* networks in \mathbb{T} by a (*c*, *s*)-bounded reduction, where *c* and *s* depend only on *k*.

Depending on the complexity of the networks in \mathbb{T} , finding a suitable (c, s)bounded reduction will in general still involve manual algorithm design. Similar to famous results about linear time algorithms for bounded treewidth [11], our proofs just guarantee the existence of small reductions.

This paper is organized as follows: this section concludes with related work. In Section 2, we describe the system model in detail. Section 3 contains the main results of the paper. Section 4 describes the impossibility of cut-offs for $CTL \setminus X$. Finally, the conclusion in Section 5 briefly considers further performance enhancements for practical applications of our method.

Related Work. Verification of parameterized systems is well known to be undecidable [2,25]. Many interesting approaches to this problem have been developed over the years, including the use of symbolic automata-based techniques [22,6, 26,7,1,4], network invariants [3,24], predicate abstraction [23], or symmetry [10,16,20,18,19]. In [5], cut-offs were used for the verification of systems sharing common resources, where the access to the resources is managed according to a FIFO-based policy.

In addition to [17] mentioned above, Emerson et al. have shown a large number of fundamental results involving cut-offs. The paper [13] by Emerson and Kahlon also considers LTL\X cut-offs for arbitrary network topologies with multiple tokens, but each of them is *confined to two processes* which renders their model incomparable to ours. Other previous work by Emerson and Kahlon [12, 15,14] consider other restricted forms of process interaction. [21] considers the verification of single index properties for systems with multiple synchronous processes.

Indexed temporal logic was introduced in [9]. This paper also considers identical processes arranged in ring topology.

The work that is closest in spirit to our negative results on $CTL^* \setminus X$ logic is the work by Browne, Clarke and Grumberg in [8] which shows how to characterize Kripke structures up to bisimilarity using fragments of CTL^* . Our results show that even $CTL^* \setminus X$ with only two atomic propositions is sufficient to describe an infinite class of Kripke structures which are not bisimilar to each other. In other words, bisimilarity over the class of Kripke structures with two labels gives rise to an infinite number of equivalence classes.

2 Computation Model

Network Topologies. A *network graph* is a finite directed graph G = (S, C) without self-loops, where S is the set of *sites*, and C is the set of *connections*. Without loss of generality we assume that the sites are numbers, i.e., $S = \{1, 2, ..., |S|\}$. A (network) *topology* \mathbb{T} is a class of network graphs.

Token Passing Process. A single token passing process *P* (process) is a *labeled transition system* (Q, Σ, δ, I) such that:

- $-Q = \hat{Q} \times B$, where \hat{Q} is a finite, nonempty set and $B = \{0,1\}$. Elements of Q will be called *local states*. The boolean component of a local state indicates the possession of the token. We say that a local state (q, b) holds the token if b = 1.
- $\Sigma = \Sigma_f \cup \Sigma_d \cup \{\text{rcv}, \text{snd}\}\$ is the set of actions. The actions in Σ_d are token dependent actions, those of Σ_f are called token independent actions, and $\{\text{rcv}, \text{snd}\}\$ are actions to receive and send the token. The sets Σ_f , Σ_d are mutually exclusive.

- $-\delta \subseteq Q \times \Sigma \times Q$ is a transition relation, such that every $((q, b), a, (q', b')) \in \delta$ fulfills the following conditions:
 - (a) A free transition does not change token possession: $a \in \Sigma_f \Rightarrow b = b'$
 - (b) A dependent transition can execute only if the process possesses the token: $a \in \Sigma_d \Rightarrow b = b' = 1$
 - (c) A receive establishes possession of token: $a = rcv \Rightarrow b = 0, b' = 1$
 - (d) A send revokes the possession of token: $a = \text{snd} \Rightarrow b = 1, b' = 0$
- $-I \subseteq Q$ is the set of initial states.

Topological Composition. Let G = (S, C) be a network graph and P = (Q, Σ, δ, I) be a single token process. Then P^G denotes the concurrent system containing n = |S| instances of P denoted by $P_s, s \in S$. The only synchronization mechanism between the processes is the passage of a token according to the network graph G. Formally, the system P^G is associated with a transition system $(\mathcal{Q}, \Delta, \mathcal{I})$ defined as follows:

- $\mathcal{Q} = \{ (q_1, \dots, q_n) \in Q^n \mid \text{exactly one of the } q_i \text{ holds the token} \}. \\ \mathcal{\Delta} \subseteq \mathcal{Q}^{2n} \text{ is defined as follows: a transition } (q_1, q_2, \dots, q_n) \to (q'_1, q'_2, \dots, q'_n)$ is in Δ in one of two cases:
 - (a) Asynchronous Transition: there exist an index $j \in \{1, ..., n\}$ and an action $a \in \Sigma_f \cup \Sigma_d$ such that $(q_j, a, q'_j) \in \delta$, and for all indices $i \neq j$ we have $q_i = q'_i$. In other words, only process P_j makes a transition (different from a send or receive).
 - (b) Token Transition: there exist a network connection $(j,k) \in C$ in the network graph, such that $(q_i, \text{snd}, q'_i) \in \delta$, $(q_k, \text{rev}, q'_k) \in \delta$, and $q_i = q'_i$ for all indices i different from j, k.
- $-\mathcal{I} = \{(q_1, \ldots, q_n) \in I^n \mid \text{exactly one of the } q_i \text{ holds the token}\}.$

An *execution path* is considered fair if and only if every process P_i receives and sends the token infinitely often. We assume that every system P^{G} that we consider has fair paths. An immediate consequence of the fairness condition is that a system $P^{\hat{G}}$ can have fair paths only if G is strongly connected.

We shall use indexed temporal logics, which can refer explicitly to the atomic propositions of each process P_i , to specify properties of the compound systems. For each local state q in Q we introduce propositional variables $q(1), \ldots, q(n)$. The atomic proposition q(i) says that process P_i is in state q. Thus, for a global state q we define

> $q \models q(i)$ iff in global state g, process P_i is in state q.

Starting from this definition for atomic propositions, we can easily define common temporal logics such as CTL or LTL in a canonical way. Throughout this paper, we will assume that the path quantifiers A and E quantify over fair paths. Further we assume that LTL formulas are implicitly quantified by E. This restriction simplifies our proofs but does not restrict generality.

Example 1. The formula $\mathbf{G}(q(1) \Rightarrow \mathbf{F}q(2))$ says that whenever process P_1 is in state q then process P_2 will be in state q sometime in the future.

For increased expressibility we permit that in an atomic formula q(x) the process index x is a variable (called *index variable*) which can take any value from 1 to |S|, the total number of processes. Thus, x can refer to arbitrary processes. We shall write $\varphi(x_1, \ldots, x_n)$ to indicate that the temporal formula φ depends on the index variables x_1, \ldots, x_n . We can substitute the index variables in a formula $\varphi(x_1, \ldots, x_k)$ by integer values i_1, \ldots, i_k in the natural way, and denote the resulting formula by $\varphi(i_1, \ldots, i_k)$.

In addition to substitution by constants, we can also quantify over the index variables $x_1, \ldots x_n$ using a prefix of existential and universal quantifiers with the natural semantics. Such formulas are called quantified temporal formulas. For example, the formula $\forall x \exists y. \varphi(x, y)$ means "For all processes x there exists a process y, such that the temporal formula $\varphi(x, y)$ holds." A formula without quantifier prefix is called *quantifier-free*. If all index variables in a formula are bound by quantifiers we say that the formula is *closed*, and *open* otherwise. The quantifier-free part of a quantified formula is called the *matrix* of a formula.

Example 2. The formula $\exists x, y. \mathbf{G}(q(x) \Rightarrow \mathbf{F}q(y))$ says that there exist two processes P_x and P_y , such that whenever process P_x is in state q then process P_y will be in state q some time in future.

The formal semantics of this logic is straightforward and is omitted for the sake of brevity.

Definition 1 (k-Indexed Temporal Formula). Let \mathcal{L} be a temporal logic. A *k-indexed* temporal formula is a formula whose matrix refers to at most k different processes, i.e., there are at most k different constant indices and index variables.

3 Reductions for Indexed LTL\X Specifications

In this section, we will show how to reduce the model checking question $P^G \models \varphi$ to a series of model checking questions on smaller systems P^{G_i} 's where we can bound the size of the network graphs G_i as well as the number of the G_i 's. For the sake of simplicity, we will start with the special case of 2-indexed existential LTL\X specifications, which can be readily generalized to the full case.

3.1 Existential 2-Indexed LTL \ X Specifications

In this section we show how to verify simple 2-indexed LTL\X properties of the form $\exists i, j.\varphi(i, j)$, where $i \neq j$. We will use the combinatorial insights we obtain from this case to obtain the more general results later on.

Recall that 2-indexed properties are concerned only with properties of two processes in a given system. Our process communication model implies that two processes P_i and P_j can only affect each other by passing or receiving a token. Consequently, the synchronization between P_i and P_j crucially depends on the paths between sites i and j in the network graph. The following example is crucial to understanding the intuition behind our approach:

Example 3. The Figure below shows one path $\pi = i, a, b, i, j, b, c, i, c, j, ...$ in a network graph.



Suppose that we are only interested in properties concerning the processes P_i and P_j , but not in processes P_a, P_b, P_c . Then only the sequence of the *i*'s and *j*'s in the path are of interest. Looking at π from left to right, we see four possibilities for what can happen between *i* and *j*: (1) P_i sends a token, and receives it back without P_j seeing it (formally, we will write $\Phi_{\bigcirc}(i, j)$ to denote this); (2) P_i passes the token directly to $P_j(\Phi_{\rightarrow}(i, j))$; (3) P_j sends the token to P_i through several intermediate sites $(\Phi_{\rightarrow}(j, i))$; and (4) P_i sends the token back to P_j through several intermediate sites $(\Phi_{\rightarrow}(i, j))$. There are two more possibilities which do not occur in π : (5) $\Phi_{\rightarrow}(j, i)$ and (6) $\Phi_{\bigcirc}(j, i)$. The important insight is the following: If we know which of these 6 cases can occur in a network graph G, then we have all information needed to reason about the communication between P_i and P_j .

We will later construct small network graphs with 4 nodes where the sites i and j are represented by two distinguished nodes $site_1$ and $site_2$, while all other sites are represented by two "hub" nodes hub_1 and hub_2 .

This example motivates the following definitions:

Definition 2 (Free Path). Let I be a set of indices, and π be a path in a network graph G. We say that π is I-free, if π does not contain a site from I.

We now define three kinds of path types which will be shown to capture all relevant token paths between two processes P_i and P_j .

Definition 3 (Connectivity, Characteristic Vectors). Let i, j be indices in a network graph G. We define three connectivity properties of the indices i, j:

$G\models \varPhi_{\circlearrowleft}(i,j)$	"There is a $\{j\}$ -free path from i to itself."
$G \models \Phi_{\leadsto}(i,j)$	"There is a path from i to j via a third node not in $\{i, j\}$."
$G \models \Phi_{\rightarrow}(i,j)$	"There is a direct edge from <i>i</i> to <i>j</i> ."

Using the connectivity properties, we define an equivalence relation \sim_2 on network graphs: Given two network graphs G_1 and G_2 along with two pairs of indices a_1, b_1 and a_2, b_2 , we define

$$(G_1, a_1, b_1) \sim_2 (G_2, a_2, b_2)$$

iff for every $\Phi \in \{\Phi_{\circlearrowright}, \Phi_{\leadsto}, \Phi_{\rightarrow}\},\$

$$G_1 \models \Phi(a_1, b_1) \iff G_2 \models \Phi(a_2, b_2)$$
 and
 $G_1 \models \Phi(b_1, a_1) \iff G_2 \models \Phi(b_2, a_2)$

If $(G_1, a_1, b_1) \sim_2 (G_2, a_2, b_2)$ we say that the indices a_1, b_1 in G_1 have the same connectivity as the indices a_2, b_2 in G_2 .

The characteristic vector $v(G_1, a_1, b_1)$ is the 6-tuple containing the truth values of $G_1 \models \Phi_{\bigcirc}(a_1, b_1)$, $G_1 \models \Phi_{\rightarrow}(a_1, b_1)$, $G_1 \models \Phi_{\rightarrow}(a_1, b_1)$, $G_1 \models \Phi_{\bigcirc}(b_1, a_1)$, $G_1 \models \Phi_{\rightarrow}(b_1, a_1)$, and $G_1 \models \Phi_{\rightarrow}(b_1, a_1)$.

By definition it holds that $(G_1, a_1, b_1) \sim_2 (G_2, a_2, b_2)$ iff they have the same characteristic vectors, i.e., $v(G_1, a_1, b_1) = v(G_2, a_2, b_2)$. Since the number of characteristic vectors is constant, it follows that \sim_2 has finite index. The characteristic vectors can be viewed as representatives of the equivalence classes.



Fig. 1. Network Graphs A, B, realizing two different characteristic vectors

Example 4. Consider the network graphs A, B of Figure 1. It is easy to see that $(A, site_1, site_2)$ has characteristic vector (1,1,1,1,1,1), i.e.,

$$A \models \Phi_{\bigcirc}(site_1, site_2) \land \Phi_{\leadsto}(site_1, site_2) \land \Phi_{\rightarrow}(site_1, site_2) \land \Phi_{\bigcirc}(site_2, site_1) \land \Phi_{\rightarrow}(site_2, site_1) \land \Phi_{\rightarrow}(site_2, site_1)$$

and $(B, site_1, site_2)$ has characteristic vector (0, 1, 0, 1, 1, 0), i.e.,

$$\begin{split} B &\models \neg \Phi_{\circlearrowright}(site_1, site_2) \land \Phi_{\leadsto}(site_1, site_2) \land \neg \Phi_{\rightarrow}(site_1, site_2) \land \\ \Phi_{\circlearrowright}(site_2, site_1) \land \Phi_{\leadsto}(site_2, site_1) \land \neg \Phi_{\rightarrow}(site_2, site_1). \end{split}$$

Note that a network graph will in general have several characteristic vectors depending on the indices we consider. The set of characteristic vectors of a graph G can be efficiently computed from G in quadratic time. The crucial insight in our proof is that for two processes P_i and P_j , the connectivity between their indices i, j in the network graph determines the satisfaction of quantifier-free LTL\X properties $\varphi(i, j)$ over P^G :

Lemma 1 (2-Index Reduction Lemma). Let G_1, G_2 be network graphs, P a process, and $\varphi(x, y)$ a 2-indexed quantifier-free LTLX property. Let a_1, b_1 be a pair of indices on G_1 , and a_2, b_2 a pair of indices on G_2 . The following are equivalent:

(a) $(G_1, a_1, b_1) \sim_2 (G_2, a_2, b_2)$, i.e., a_1, b_1 and a_2, b_2 have the same connectivity. (b) $P^{G_1} \models \varphi(a_1, b_1)$ iff $P^{G_2} \models \varphi(a_2, b_2)$. The lemma motivates the following model checking strategy: Given a (possibly complicated) network graph G_1 and two of its sites i, j, we can try to obtain a simpler network $G_2 := G_{(i,j)}$, with two special nodes $site_1$ and $site_2$ that have the same connectivity in G_2 as the indices i and j in G_1 , and thus satisfies condition (a) of the lemma. For the case of two indices, we can always find such a network graph $G_{(i,j)}$ with at most 4 sites.

Proposition 1. For each graph G and indices i, j there exists a 4-node graph $G_{(i,j)}$ called the connection topology of i, j, having two special sites site₁ and site₂ such that

 $(G, i, j) \sim_2 (G_{(i,j)}, site_1, site_2).$

In other words, the indices i and j in G have the same connectivity as the indices site₁ and site₂ in $G_{(i,j)}$.

Since $G_{(i,j)}$ satisfies condition (a) of Lemma 1, we obtain the following important consequence:

Corollary 1. Let $\varphi(i, j)$ be a 2-indexed quantifier-free LTLX property. Then

 $P^G \models \varphi(i,j)$ iff $P^{G_{(i,j)}} \models \varphi(site_1, site_2).$

Thus, we have achieved a reduction from a potentially large network graph G to a 4-node network graph $G_{(i,j)}$. We will now show how to actually construct the connection topology $G_{(i,j)}$.

Construction of $G_{(i,j)}$. We construct the reduction graphs as follows, $G_{(i,j)}$ has four sites: $site_1, site_2, hub_1$, and hub_2 . The sites $site_1$ and $site_2$ are called *primary sites*. They represent the sites of interest *i* and *j*. The other sites are called *hubs*, and they represent the other nodes of the graph *G*. Let us describe in more detail the role of these different nodes. Recall that to satisfy Proposition 1, the sites $site_1$ and $site_2$ in $G_{(i,j)}$ should have the same connectivity as *i*, *j* in *G*. Therefore:

- If $\Phi_{\rightarrow}(i,j)$ holds in G (i.e., there exists a path from *i* to *j* in G that goes through a third node), then $\Phi_{\rightarrow}(site_1, site_2)$ has also to hold in $G_{(i,j)}$, i.e., there should exist in $G_{(i,j)}$ a path from $site_1$ to $site_2$ that goes through a third node. The site hub_1 will play the role of this "third node". Therefore, in this case, $G_{(i,j)}$ contains an edge from $site_1$ to hub_1 , and from hub_1 to $site_2$.
- In the same manner, if $\Phi_{\bigcirc}(i, j)$ holds in G (i.e., there exists a path from *i* to itself in G that does not go through *j*), then $\Phi_{\bigcirc}(site_1, site_2)$ should also be true in $G_{(i,j)}$. As previously, this is ensured by considering the following edges: $(site_1, hub_1)$ and $(hub_1, site_1)$.
- Finally, if $\Phi_{\rightarrow}(i,j)$ holds in G (i.e., there exists a direct edge in G from i to j), then $G_{(i,j)}$ should also contain the edge (*site*₁, *site*₂).
- The paths from j to i are treated in a symmetrical way.

For example, let *H* be a graph having as sites i, j, k, and *l* (among others), such that v(H, i, j) = (1, 1, 1, 1, 1, 1), and v(H, k, l) = (0, 1, 0, 1, 1, 0); then the graphs *A* and *B* of Example 4 correspond respectively to the reduction graphs $H_{(i,j)}$ and $H_{(k,l)}$.

Since our fairness assumption implies that the network is strongly connected, not all characteristic vectors actually occur in practice. A closer analysis yields the following bound:

Proposition 2. For 2 indices, there exist at most 36 connection topologies.

Proof. By our fairness assumption, every connection topology must be strongly connected. This implies that the following conditions must hold:

- At least one of $\Phi_{\rightarrow}(i,j)$ or $\Phi_{\rightarrow}(i,j)$ must be true.
- At least one of $\Phi_{\rightarrow}(j,i)$ or $\Phi_{\rightarrow}(j,i)$ must be true.

Consequently a detailed counting shows that the number of different possible characteristic vectors is $3 \times 3 \times 4 = 36$.

Let us now return to the question of verifying properties of the form $\exists x, y.\varphi(x, y)$. Note that Corollary 1 only provides us with a way to verify one quantifier-free formula $\varphi(i, j)$. Given a system P^G , we define its 2-topology, denoted by $T_2(G)$, as the collection of all different connection topologies appearing in G. Formally,

Definition 4. Given a network graph G = (S, C) the 2-topology of G is given by

$$T_2(G) = \{G_{(i,j)} \mid i, j \in S, i \neq j\}.$$

By Proposition 2, we know that $|T_2(G)| \leq 36$. Since we can express $\exists x, y.\varphi(x,y)$ as a disjunction $\bigvee_{i,j\in S} \varphi(i,j)$ we obtain the following result as a consequence of Corollary 1:

Theorem 1. The following are equivalent:

(i) $P^G \models \exists x, y.\varphi(x, y)$

(ii) There exists a connection topology $T \in T_2(G)$, such that $P^T \models \varphi(site_1, site_2)$.

Thus, we obtain the following reduction algorithm for model checking $P^G \models \exists x, y.\varphi(x, y)$:

- 1: Determine $T_2(G)$.
- 2: For each $T \in T_2(G)$, model check $P^T \models \varphi(site_1, site_2)$.
- 3: If one of the model checking calls is successful then output "true" else output "false".

3.2 Existential k-Indexed LTL\X Specifications

We will now show how to generalize the results of the previous section to k-indexed properties. Throughout this section, we will write expressions such as \overline{i} to denote k-tuples of indices, and \overline{x} to denote k-tuples of variables. We will first adapt the notion of connectivity as follows. Let $\overline{i} = i_1, i_2 \dots i_k$ be a sequence of indices, and $I = \{i_1, i_2 \dots i_k\}$. Then we define the following connectivity properties:

$G \models \Phi_{\circlearrowright}(x, I)$	"There is an $(I \setminus \{x\})$ -free path from x to itself."
$G \models \Phi_{\leadsto}(x, y, I)$	"There is a path from x to y via a third node not in I ."
$G \models \Phi_{\rightarrow}(x, y)$	"There is a direct edge from x to y ."

By instantiating the variables x and y by the indices i_1, \ldots, i_k in all possible ways, we obtain a finite number of different conditions which will describe all possible connectivities between the indices i_1, \ldots, i_k .

As in the previous section, we can define an equivalence relation \sim_k , where $(G_1, \overline{i}) \sim_k (G_2, \overline{j})$ iff the indices \overline{i} have the same connectivity in G_1 as the indices \overline{j} in G_2 . Since the number of conditions is bounded, \sim_k is an equivalence relation of finite index, and we can describe each equivalence class by a characteristic vector $v(G, \overline{v})$. Like in the previous section, we define the *k*-connection topologies, $G_{(i_1,i_2...i_k)}$ of the processes $P_{i_1}, P_{i_2} \dots P_{i_k}$ in G as the smallest graphs that preserve all the connectivity properties between the processes $P_{i_1}, P_{i_2} \dots P_{i_k}$. The construction of the topology graphs is illustrated in Figure 2.

The unfilled nodes $site_1, \ldots, site_k$ in the graph are the primary sites. There is a *hub* site associated with each primary site. Moreover, there is an edge from each hub *hub_j* back to its primary $site_j$ if there is an $(I \setminus \{i_j\})$ -free path from i_j to itself. There is an edge from *hub_j* to $site_l$ if there is a path from i_j to i_l in G via a third node not in I, and there is an edge from $site_j$ to $site_l$ if there exists a direct edge (i_j, i_l) in G.



Fig. 2. An example of a 5-index connection topology

Analogous to the bounds on 2-connection topologies it can be shown that each *k*-connection topology has at most 2k processes and that there are at most $3^{k(k-1)}2^k$ distinct *k*-connection topologies. By an argument analogous to that of the previous section, we obtain the following corollary

Corollary 2. Let $\varphi(\bar{x})$ be a *k*-indexed quantifier-free LTLX property. Then

$$P^G \models \varphi(\overline{i})$$
 iff $P^{G_{(\overline{i})}} \models \varphi(site_1, site_2, \dots, site_k).$

The notion of *k*-topology is also defined completely analogously:

Definition 5. Given a network graph G = (S, C) the k-topology of G is given by

 $T_k(G) = \{G_{(\bar{i})} \mid \bar{i} \in S^k, all indices in \bar{i} are distinct\}.$

Consequently, we obtain a model checking procedure from the following theorem, similar to the case of 2-indices:

Theorem 2. The following are equivalent:

- (i) $P^G \models \exists \bar{x}.\varphi(\bar{x})$
- (ii) There exists a connection topology $T \in T_k(G)$, such that $P^T \models \varphi(site_1, site_2, \dots, site_k)$.

As mentioned before $|T_k(G)| \leq 3^{k(k-1)}2^k$.

3.3 Specifications with General Quantifier Prefixes

In this section we will show how to obtain reductions for *k*-indexed specifications with first order prefixes.

Let us for simplicity consider the 2-indexed formula $\Phi := \forall x \exists y.\varphi(x,y)$. Over a network graph G = (S,C), |S| = n it is clear that Φ is equivalent to $\wedge_{1 \leq i \leq n} \vee_{1 \leq j \leq n} \varphi(i, j)$. A naive application of Corollary 2 would therefore require n^2 calls to the model checker which may be expensive for practical values of n. In practice, however, we can bound the number of model checker calls by $|T_2(G)|$ since this is the maximum number of *different* connection topologies. We conclude that the n^2 model checker calls must contain repetitions. In the program, we can make sure that at most 36 calls to the model checker are needed. We obtain the following algorithm:

- 1: Determine $T_2(G)$.
- 2: For each $T \in T_2(G)$
- 3: model check $P^T \models \varphi(site_1, site_2)$
- 4: g[T] := 1 iff model checking successful, and 0 otherwise
- 5: Output $\bigwedge_{1 \le i \le n} \bigvee_{1 \le j \le n} g[G_{(i,j)}]$.

By simplifying the formula in line 5, we may further increase performance. The algorithm can be adapted for k indices in the obvious way. To state the main theorem of this section, we define (c, s)-bounded reductions, where c bounds the number of calls to the model checker, and s bounds the size of the network graph.

Definition 6 ((c, s)-Bounded Reduction). Let G, P be as above, and φ a closed k-indexed formula with matrix $\varphi'(x_1, \ldots, x_k)$. Let Ψ denote a property of interest (e.g., the model checking property " $P^G \models \varphi$ "). A (c, s)-bounded reduction of property Ψ is given by:

- a sequence of c reduced network graphs $G_i = (S_i, C_i), 1 \le i \le c$ such that $|S_i| \le s$. called reduction graphs.
- a boolean function B over c variables g_1, \ldots, g_c , such that

 Ψ iff $B(g_1, \ldots, g_c) = 1$ where $g_i := 1$ iff $G_i^P \models \varphi'(site_1, \ldots, site_k)$

In other words, property Ψ is decided by c calls to the model checker, where in each call the network graph is bounded by s.

Further, we say that a class \mathcal{L} of specifications has (c, s) bounded reduction if for all network graphs G and any $\varphi \in \mathcal{L}$, the property $P^G \models \varphi$ has (c, s)-bounded reduction. We can now state our main result:

Theorem 3. Let φ be any k-indexed LTLX specification. Then the model checking problem " $P^G \models \varphi$ " has polynomial-time¹ computable $(3^{k(k-1)}2^k, 2k)$ -bounded reductions.

In fact, the sequence of reduced network graphs is just the different k-connection topologies occurring in G. This implies that given k and network graph G, all k-indexed LTL\X specifications have the same reduction. Stated another way, LTL\X has $(3^{k(k-1)}2^k, 2k)$ -bounded reduction.

3.4 Cut-Offs for Network Topologies

In this section, we prove the existence of cutoffs for network topologies, i.e., (infinite) classes of network graphs. We say that a class of network graphs has cutoff (c, s), if the question whether **all** the network graphs in this topology satisfy the specification has a (c, s)-bounded reduction.

Definition 7 (**Cut-Off**). Let \mathbb{T} be a network topology, and \mathcal{L} a class of specifications, \mathbb{T} has a cut-off (c, s) for \mathcal{L} if for all specifications $\varphi \in \mathcal{L}$ the property

$$\Psi :=$$
 " $\forall G \in \mathbb{T}$. $P^G \models \varphi$ "

has a (c, s)-bounded reduction.

It is not hard to prove that a (c, s)-bounded reduction for a network graph translates to a cut-off for a network topology:

Theorem 4. For k-indexed specifications, all network topologies \mathbb{T} have $(2k, 3^{k(k-1)}2^k)$ -bounded reductions.

Note that the theorem does not provide us with an *effective* means to find the reduction; it does however guarantee that at least in principle we can always find a cutoff by investigating the topology \mathbb{T} .

¹ In the size of the network graph G.

4 Bounded Reductions for CTL \ X Are Impossible

In this section, we show that indexed CTL\ X formulas over two indices don't have (c, s)-bounded reductions. We will first show the following generic result about CTL\ X:

Theorem 5. For each number *i* there exists an CTL\ X formula φ_i with the following properties:

 φ_i is satisfiable (and has a finite model). φ_i uses only two atomic propositions l and r. Every Kripke structure K where φ_i is true has at least i states. φ_i has the form $\mathbf{EF}\varphi'_i$.

The result is true even when the Kripke structure is required to have a strongly connected transition relation.

Proof. Our goal is to describe a formula φ_i using atomic propositions l and r whose models must have at least i states. We will construct a large conjunction $\bigwedge_{\psi \in \Gamma} \psi$, and describe which formulas to put in Γ . The idea is simple: Γ needs to contain i CTL\X formulas which describe the existence of i different states. Then the formula **EF** $\bigwedge_{\psi \in \Gamma} \psi$ will be the sought for φ_i .



Fig. 3. The Kripke structure K, constructed for three levels. The dashed lines indicate the connections necessary to achieve a strongly connected graph

Consider a Kripke structure *K* as in Figure 3:

- In Level 0, it contains two distinct states L, R labelled with l and r respectively. To express the presence of these states, we include the formulas, let $\psi_0^1 := (l \wedge \neg r)$ and $\psi_0^2 := (r \wedge \neg l)$, and include $\mathbf{EF}\psi_0^1$ and $\mathbf{EF}\psi_0^2$ into Γ . It is clear that $\mathbf{EF}\psi_0^1$ and $\mathbf{EF}\psi_0^2$ express the presence of two mutually exclusive states.
- In Level 1, K contains $2^2 1 = 3$ states, such that the first one has $\{L, R\}$ free paths to L and R, the second one an $\{L, R\}$ -free path only to L, and the third one an $\{L, R\}$ -free path only to R. The characteristic properties of level 1 states are expressed by formulas

$$\psi_1^1 := \mathbf{E}\mathbf{F}^-\psi_0^1 \wedge \mathbf{E}\mathbf{F}^-\psi_0^2$$

$$\psi_1^2 := \mathbf{E}\mathbf{F}^-\psi_0^1 \wedge
eg \mathbf{E}\mathbf{F}^-\psi_0^2$$

 $\psi_1^{\mathbf{i}} := \mathbf{E}\mathbf{F} \quad \psi_0^{\mathbf{i}} \land \neg \mathbf{E}\mathbf{F} \quad \psi_0^{\mathbf{i}} \\ \psi_1^{\mathbf{3}} := \neg \mathbf{E}\mathbf{F}^- \psi_0^{\mathbf{i}} \land \mathbf{E}\mathbf{F}^- \psi_0^{\mathbf{2}}$

where $\mathbf{EF}^{-}x$ denotes $\mathbf{E}(\neg l \land \neg r)\mathbf{U}x$, i.e., a variant of EF which forbids paths through L and R. To enforce the existence of the Level 1 states in the Kripke structure, we include $\mathbf{EF}\psi_1^1$, $\mathbf{EF}\psi_1^2$, $\mathbf{EF}\psi_1^3$ into Γ . - In general, each Level k has at least $2^{k+1} - 1$ states which differ in their

relationship to the states in Level k - 1. The presence of such states is expressed by formulas $\mathbf{EF}\psi_{\mathbf{k}}^{\mathbf{x}}$.

All these formulas are included into Γ until the requested number *i* of different states is reached. By construction, all properties required in the theorem statement are trivially fulfilled. In particular, Figure 3 demonstrates that there always exists a strongly connected model.

Remark 1. This result is closely related to early results about characterizing Kripke structures up to bisimulation in [8]. The results in [8] give rise to the following proof idea for Theorem 5: Let K_1, \ldots, K_n be all Kripke structures with 2 labels of size $\leq i$, and let f_1, \ldots, f_n be CTL\ X formulas which characterize them up to stuttering bisimulation. Consider now the formula $\varphi_i := \bigwedge_{1 \le j \le n} \neg f_j$. By construction every model of φ_i must have > i states. At this point, however, the proof breaks down, because we do not know from the construction if φ_i is satisfiable at all. The natural way to show that φ_i has a model would be to prove that stuttering bisimulation over a 2-symbol alphabet has infinite index. This property however is a corollary to Theorem 5, and we are not aware of a proof in the literature.

For properties involving only the presence of the token, a system P^{G} , where G = (S, C) essentially behaves like a Kripke structure with set of states S and transition relation C. The proof of this assertion is not given here.

Now we can show by contradiction that indexed CTL\ X cannot have bounded reductions. Suppose CTLX did have (c, s)-bounded reduction for some s. Then, by Theorem 5, we can always find a CTL\X formula Φ such that the network graph underlying any system that satisfies Φ must have size at least c + 1. Thus CTL\X does not have bounded reductions. Consequently, we also have the following corollary:

Corollary 3. There exists a network topology \mathbb{T} for which 2-indexed CTL\ X does not have cut-offs.

5 Conclusion and Future Work

In this paper, we have described a systematic approach for reducing the verification of large and parameterized systems to the verification of a sequence of much smaller systems. The current paper is primarily concerned with the algorithmic and logical concepts underlying our approach. We will conclude this paper with further considerations concerning the practical complexity of model checking.

For simplicity, let us again consider the case of 2-indexed properties. Suppose the processes P in our network have state space |Q|. Then our reduction requires to model check up to 36 network graphs with 4 sites, resulting in a state space of $|Q|^4$. Even this model checking problem may be expensive in practice. By a close analysis of our proofs, it is however possible to reduce the state space even further to $O(|Q|^2)$.

It is easy to show that Lemma 1 will hold even when the *processes at the* hubs are simple dummy processes containing two states whose mere task is to send and receive the token infinitely often. Consequently, the systems $P^{G_{(i,j)}}$ will have state space of size $2^2 \times |Q|^2$.

The results in this paper on LTL\X were derived assuming fairness condition on the systems. We can obtain similar reductions by removing this assumption. Doing away with fairness necessitates the consideration of two more path types other than the ones described in Section 3.1. Consequently, the topology graphs have more than 4 sites and also the number of different topology graphs increases. Reductions in non-fair case will be described in a future work.

References

- P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Regular model-checking made simple and efficient. In *In Proceedings 13th International Conference on Concurrency Theory (CONCUR)*, volume 2421 of *Lecture Notes in Computer Science*, pages 116–130. Springer-Verlag, 2002.
- 2. K. Apt and D. Kozen. Limits for automatic verification of finite state concurrent systems. *Information Processing Letters*, 15:307–309, 1986.
- 3. T. Arons, A. Pnueli, S. Ruah, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *Proc. 13th Intl. Conf. Computer Aided Verification (CAV)*, 2001.
- B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large. In 15th Intern. Conf. on Computer Aided Verification (CAV'03). LNCS, Springer-Verlag, 2003.
- A. Bouajjani, P. Habermehl, and T. Vojnar. Verification of Parametric Concurrent Systems with Prioritized FIFO Resource Management. In *Proceedings of CONCUR'03*, 2003.

- A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In 12th Intern. Conf. on Computer Aided Verification (CAV'00). LNCS, Springer-Verlag, 2000.
- 7. A. Bouajjani and T. Touili. Extrapolating tree transformations. In 14th Intern. Conf. on Computer Aided Verification (CAV'02). LNCS, Springer-Verlag, 2002.
- M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.
- 9. M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 81:13–31, 1989.
- 10. E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal model checking. In Proc. 5th Intl. Conf. Computer Aided Verification (CAV), 1993.
- 11. B. Courcelle. Graph rewriting: An algebraic and logic approach. B:459-492, 1990.
- 12. A. E. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *17th International Conference on Automated Deduction*, pages 236–254, 2000.
- 13. A. E. Emerson and V. Kahlon. Model checking larage-scale and parameterized resource allocation systems. In *Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 251–265, 2002.
- A. E. Emerson and V. Kahlon. Model checking guarded protocols. In *Eighteenth* Annual IEEE Symposium on Logic in Computer Science (LICS), pages 361–370, 2003.
- 15. A. E. Emerson and V. Kahlon. Rapid parameterized model checking of snoopy cache protocols. In *Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 144–159, 2003.
- 16. E. A. Emerson, J. Havlicek, and R. Trefler. Virtual symmetry. In LICS, 2000.
- 17. E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In ACM Symposium on Principles of Programming Languages (POPL'95), 1995.
- 18. E. A. Emerson and A. Sistla. Utlizing symmetry when model-checking under fairness assumptions: An automata theoretic approach. *TOPLAS*, 4, 1997.
- 19. E. A. Emerson and A. P. Sistla. Symmetry and model checking. In *Proc. 5th Intl. Conf. Computer Aided Verification (CAV)*, 1993.
- 20. E. A. Emerson and R. Trefler. From asymmetry to full symmetry. In *CHARME*, 1999.
- 21. S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of ACM*, 39, 1992.
- Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proc. CAV'97*, volume 1254 of *LNCS*, pages 424–435. Springer, June 1997.
- 23. S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification". In *Proc. CAV'04*. To appear.
- 24. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Lecture Notes in Computer Science*, 2001.
- 25. I. Suzuki. Proving properties of a ring of finite state machines. *Information Processing Letters*, 28:213–214, 1988.
- 26. T. Touili. Widening Techniques for Regular Model Checking. In *1st vepas work-shop*. Volume 50 of Electronic Notes in Theoretical Computer Science, 2001.

Reversible Communicating Systems

Vincent Danos¹* and Jean Krivine²

¹ Université Paris 7 & CNRS ² INRIA Rocquencourt

Abstract. One obtains in this paper a process algebra RCCS, in the style of CCS, where processes can backtrack. Backtrack, just as plain forward computation, is seen as a synchronization and incurs no additional cost on the communication structure. It is shown that, given a past, a computation step can be taken back if and only if it leads to a causally equivalent past.

1 Introduction

Backtracking means rewinding one's computation trace. In a distributed setting, actions are taken by different threads of computation, and no currently running thread will retain a complete description of the others past. Therefore, there is no guarantee that when a given thread goes back in its own local computation history, this will correspond to going back a step in the global computation trace. Of course, one could ask a thread willing to go back a step, to first verify that it was the last to take an action. But then all concurrent behaviour would be lost, not speaking about the additional communication machinery this choice would incur. On the other hand, letting any thread freely backtrack would result in losing the initial computation structure and reaching computation states which were formerly inaccessible. So, one has to strike a compromise here.

This is what we propose in this paper. A notion of distributed backtracking built on top of Milner's CCS [1] is provided. At any time, a thread may either fork or synchronize with another thread, and in both cases, the action taken is recorded in a memory. When the thread wants to rewind a computation step, it has to synchronize with either its sibling, in the case the last action was a fork, or with its synchronization partner in the case the last action was a synchronization. Thus backtrack is considered also as a synchronization mechanism.

This mechanism can be construed as a distributed monitoring system and it meshes well with the specifics of the host calculus CCS. Backtrack doesn't involve any additional communication structure and we could obtain a syntax, termed RCCS, for *reversible* CCS, that stays really close to ordinary CCS.

There is another aspect in which the syntax seems to do well. The compromise it corresponds to, has a clear-cut theoretical characterization. Given a process

^{*} Corresponding author: Équipe PPS, Université Paris 7 Denis Diderot, Case 7014, 2 Place Jussieu 75251 PARIS Cedex 05, Vincent.Danos@pps.jussieu.fr

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 292-307, 2004.

[©] Springer-Verlag Berlin Heidelberg 2004

and a past, one can show that the calculus allows backtrack along any *causally equivalent* past. Computation traces originating from a process are said to be causally equivalent when one can transform one in the other by commuting successive concurrent actions, or cancelling successive inverse actions.

A similar notion of computation trace equivalence exists in λ -calculus which Lévy could characterize by a suitable labelling system [2,3]. Thus, a pretty good summary of the theoretical status of this backtracking mechanism, is to say that RCCS is a Lévy labelling for CCS. Two reduction paths will be equivalent if and only if they lead to the same process in RCCS. This is what we prove and it seems to be the best one can expect on the theoretical side.¹

To summarize the contribution, the present study proposes a syntax for reversible communicating systems, together with a characterization, in terms of causally equivalent traces, of the exact amount of flexibility one allows in backtracking. One also explains how irreversible, or unbacktrackable actions, can be included in the picture and a procedure of memory cleansing is introduced and proved to be sound.

Following Regev [4,5], process algebras have been investigated recently for modeling biological systems. Since reversibility is the rule in biological interaction, the second author was naturally prompted to look for a theoretical setup for distributed and reversible computations. Biological modeling in a former version of RCCS was explored [6]. By that time soundness (here, corollary 1) was proved directly, and the key connection to causal equivalence went unnoticed. Future work, and in particular, applications to the synthesis of sound transactional mechanisms is discussed in the conclusions.

1.1 Related Work

Process algebras with backtracking were seen early to be valuable computational objects and independently studied by Prasad [7] and later by Bergstra et al. [8]. However, both had an exception model in mind, which while providing interesting programming constructs would not have any specific theoretical structure. Another well developed line of research, partly inspired by Lévy's work on causal equivalence in λ -calculus, and partly by the need for refined non-interleaving semantics, is that of the causal analysis of distributed systems [9–16]. However, the only concern here is forward computation. Causal analysis is thought of as a static analysis method, or a theoretical measure of how concurrent a system is, and not as inducing some policy that threads should obey in order to backtrack soundly. In some sense, we present here a synthesis of these two lines of research which, to the best of our knowledge, were never brought together to interact.

1.2 Acknowledgements

The authors wish to thank the referees for their suggestions and specifically for correcting a wrong formulation of corollary 1.

¹ This crucial property can be recast in topological terms, by saying that RCCS is the universal cover of CCS.

2 RCCS

The plan to implement backtrack is to assign to each currently running thread an individual memory stack keeping track of past communications. This memory will also serve as a naming scheme and yield a unique identifier for the thread. Upon doing a forward transition, the information needed for a potential roll-back will be pushed on the memory stack.

As said briefly in the introduction, two constraints are shaping the actual syntactic solution explained below. First the notion of past built in the memories has to have some degree of flexibility. Even if one could somehow record the complete succession of events during a distributed computation and only allow backward moves in whatever precise order was taken, this would induce fake causal dependencies on backward sequences of actions. Actions which could have been taken in any order would have to be undone in the precise incidental order in which they happened. So one should not be too rigid on the exact order in which things done have to be undone.

On the other hand the notion of past should not be too flexible. Because if it is, then one might be in danger of losing *soundness*, in that some backtracking computations could give access to formerly unreachable states. Clearly, if actions are undone before whatever action they caused is, the result is not going to be consistent.

It turns out that the solution proposed here is at the same time consistent and maximally flexible. The final take on this will be a theorem proving that any two computation traces starting in a same state and reaching a same end state are causally equivalent, or in other words that one can be rearranged so as to obtain the other by commuting concurrent actions. Consistency will follow.

But, first of all we need a syntax to describe our processes and this is the matter to which we turn in the next subsection.

2.1 A Syntax for Backtrack

Simple Processes. Simple processes are taken from CCS [1]:

Actions: $\alpha ::= a \mid \bar{a} \mid \dots$ Action on a channel
 $\mid \tau$ Processes:P ::= 0End of process $\mid \sum \alpha_i . P_i$ Guarded Choice
 $\mid (P \mid P)$ Fork
 $\mid (a)P$

Let us briefly remind that interaction consists only of binary synchronized communication. In a CCS system something happens when two processes are performing complementary actions at the same time, very much as a handshake. Recursive definitions can be dealt with, but they are not central to the point being made in this paper and we will do without them. As the notation for choice suggests, the order in which choices add up is irrelevant. Simple processes will therefore be considered only up to *additive structural congruence*, that is to say the equivalence relation generated by the following usual identities:

$$P + 0 \equiv P P_1 + P_2 \equiv P_2 + P_1 (P_1 + P_2) + P_3 \equiv P_1 + (P_2 + P_3),$$

where P_1 , P_2 , and P_3 represent processes of the guarded choice type.²

Monitored Processes. In RCCS, simple processes are not runnable as such, only monitored processes are. This second kind of process is defined as follows:

Memories:	$m::=\langle angle$	Empty memory
	$\langle 1 angle \cdot m$	Left-Fork
	$\langle 2 angle \cdot m$	Right-Fork
	$\langle *, lpha, P angle \cdot m$	Semi-Synch
	$\langle m, \alpha, P \rangle \cdot m$	Synch
Monitored Processes:	$R::=m\triangleright P$	Threads
	$ (R \mid R)$	Product
	(a)R	Restriction
Monitored Processes:	$R ::= m \triangleright P$ $ (R R)$ $ (a)R$	Threa Produ Restrictio

To sort visually our two kinds of processes, the simple ones will be ranged over by P, Q, ... while the monitored ones will be ranged over by R, S, ...

Sometimes, when it is clear from the context which kind of process is being talked about, we will say simply process in place of monitored process.

As one may readily see from the syntax definition, a monitored process can be uniquely constructed from a set of terms of the form $m \triangleright P$, which we will call its *threads*. In a thread $m \triangleright P$, *m* represents a memory carrying the information that this process will need in case it wants to backtrack. That memory is organized as a stack with the last action taken by the thread sitting on the top together with additional information that we will comment on later. There is an evident *prefix ordering* between memories which will be written \leq .

As an example we can consider the following monitored process:

$$R = \langle 1 \rangle \langle *, a, 0 \rangle \triangleright b. \overline{c}. 0 \mid \langle 2 \rangle \langle *, a, 0 \rangle \triangleright c. 0$$

It consists of two threads, namely $S_1 = \langle 1 \rangle \langle *, a, 0 \rangle \triangleright b.\overline{c}.0$ and $S_2 = \langle 2 \rangle \langle *, a, 0 \rangle \triangleright c.0$. Taking a closer look at S_1 , we see a fork action $\langle 1 \rangle$ sitting on top of its memory stack, indicating that the last interaction the thread took part in was a fork. Below one finds $\langle *, a, 0 \rangle$ indicating that the penultimate action was an *a* action exchanged with an unidentified partner *. That part of the past of S_1 is shared by S_2 as well. Actually, they both can be obtained from a same process $\langle \rangle \triangleright a.(b.\overline{c}.0 | c.0)$ as will become evident when we have a precise notion of computation.

² General sums are not allowed in the syntax; here as in the following, all sums will be supposed guarded.

Coherent Processes. Not all monitored processes are going to be of interest. Since memories are also serving as a naming scheme for threads, they should better be unique. Actually we can ask for a little more than memory uniqueness and this is what we call *coherence* and proceed now to define.

Definition 1. Coherence, written \frown , is the smallest symmetric relation such that:

 $\begin{array}{l} -\forall i, \ j: i \neq j \Rightarrow \langle i \rangle \cdot m \frown \langle j \rangle \cdot m; \\ -\forall m_1, \ m_2: m \frown m' \Rightarrow m_1 \cdot m \frown m_2 \cdot m'. \end{array}$

Memories are coherent if they branch on a fork.

Definition 2. A monitored process is said to be coherent if its memories are pairwise coherent.

Coherence implies in particular that memories are *unique* in a coherent term, since coherence is irreflexive. But, as said, it is a bit stronger than that. For instance $\langle *, b, 0 \rangle \langle 1 \rangle \triangleright P \mid \langle *, a, 0 \rangle \langle 1 \rangle \triangleright Q$ is not coherent, even if its two memories are distinct.

Define inductively the *fork structure* $\lambda(m)$ of a memory *m*:

$$\begin{aligned} \lambda(\langle \rangle) &= \langle \rangle \\ \lambda(\langle *, a, Q \rangle \cdot m) &= \lambda(\langle m', a, Q \rangle \cdot m) = \lambda(m) \\ \lambda(\langle i \rangle \cdot m) &= \langle i \rangle \lambda(m) \end{aligned}$$

An additional coherence requirement could be that for any memory m occurring in a process R, $\lambda(m)$ is exactly the forking address of m in R, where by the forking address of m in R we mean the binary string over $\{1, 2\}$ locating the thread with memory m in R. For an example of a process violating this extra condition, consider: $\langle *, a, 0 \rangle \langle 1 \rangle \langle 1 \rangle \triangleright 0 \mid \langle 2 \rangle \triangleright 0$.

From RCCS to CCS and Back 2.2

Our calculus is clearly only a "decoration" of CCS, a decoration which can be erased by way of the forgetful map φ : RCCS \rightarrow CCS:

$$\begin{array}{l} \varphi(m \triangleright P) = P \\ \varphi(R \mid S) = \varphi(R) \mid \varphi(S) \\ \varphi((a)R) = (a)\varphi(R) \end{array}$$

Conversely one can lift any CCS process to RCCS with the map $\ell(P) = \langle \rangle \triangleright P$. One has that $\varphi \circ \ell$ is the identity but not the converse ! If we go back to our first example, we see that $\ell(\varphi(R)) = \langle \rangle \triangleright (b.\bar{c}.0 \mid c.0)$. The transformation $\ell \circ \varphi$ is blanking all memories in a monitored process.

2.3 **RCCS Structural Congruence**

We now want to extend the additive structural congruence defined earlier on simple processes, to monitored processes. The most important additional rule is the following:

$$m \triangleright (P \mid Q) \equiv (\langle 1 \rangle \cdot m \triangleright P \mid \langle 2 \rangle \cdot m \triangleright Q) \tag{1}$$

It explains how memory is distributed when a process divides in two subthreads. We see that each sub-thread inherits the father memory together with a fork number indicating which of the two sons the thread is.

Another rule we need is:

$$m \triangleright (a) P \equiv (a) m \triangleright P \tag{2}$$

Both rules have a natural left to right orientation corresponding to forward computation. Take note also that both these memory rearrangements are invisible in CCS, *e.g.*, $\varphi(m \triangleright (P \mid Q))$ is actually equal to $\varphi(\langle 1 \rangle \cdot m \triangleright P \mid \langle 2 \rangle \cdot m \triangleright Q)$.

Structural congruence on monitored processes is then obtained by combining these two identities together with additive congruence. In other words, two processes are *structurally equivalent* if related by the smallest equivalence relation generated by the identities (1), (2) above, by additive congruence identities, and closed under all syntactical constructs.

Lemma 3. Coherence is preserved by structural congruence.

The only case where memories are modified is in using identity (1) where a given m is split in $\langle 1 \rangle \cdot m$ and $\langle 2 \rangle \cdot m$. By definition $\langle 1 \rangle \cdot m \frown \langle 2 \rangle \cdot m$ and an m' is coherent with m iff it is coherent with both $\langle 1 \rangle \cdot m$ and $\langle 2 \rangle \cdot m$.

Usual identities associated with the product, such as $P \mid Q \equiv Q \mid P$ are not considered here because memories are using the actual product structure of the term to memorize the fork actions. A quotient would force the manipulation of terms up to tree isomorphisms on memories. That is possible and perhaps even interesting if one wants a more mathematical view on the calculus, but certainly results in a less convenient syntax.

2.4 Transition Rules

It remains to define a proper notion of computation. To this effect, we use a labelled transition system, LTS for short, that is to say a family of binary relations over monitored processes. Specifically, transitions are of the form:

$$R \xrightarrow{\mu:\zeta} S_{q}$$

where *R*, *S* are monitored processes, ζ is a *directed action*, that is either a forward action or a backward action, while μ is an *identifier*, that is either a memory or a pair of memories:

$$\begin{array}{ll} \zeta ::= \alpha \ | \ \alpha_* & \mbox{Directed Actions} \\ \mu ::= m \ | \ m, m & \mbox{Identifiers} \end{array}$$

Basic Rules. First we have the basic transitions concerning threads:

 $\overline{m \triangleright \alpha.P + Q} \xrightarrow{m:\alpha} \langle *, \alpha, Q \rangle \cdot m \triangleright P \qquad \langle *, \alpha, Q \rangle \cdot m \triangleright P \xrightarrow{m:\alpha} m \triangleright a.P + Q$

The first transition is a forward transition whereby the thread does an action α . A memory triplet $\langle *, \alpha, Q \rangle$ containing this action, as well as the leftover part Q is pushed on top of the memory, and the thread proceeds further down its code. The first element in the memory triplet '*' stands for an unidentified partner. The transition itself is indexed by the action α so that it can be synchronized with a transition bearing a complementary action, and the memory m, which will be used to identify oneself in the synchronization.

The second transition goes backward. The process is now undoing an action which is on the top of its memory and is therefore the last action it took. As we discussed already, many actions may have happened in the meantime in the global computation, but none originated from that particular thread, or they were undone before. Backward actions are treated on a par with forward actions and in particular, backtracking also involves communicating.

Contextual Rules. We have seen what transitions a thread may trigger. These transitions can also be done in a context:

$$\frac{R \xrightarrow{\mu:\zeta} R'}{R \mid S \xrightarrow{\mu:\zeta} R' \mid S} \qquad \frac{R \xrightarrow{\mu:\zeta} R'}{S \mid R \xrightarrow{\mu:\zeta} S \mid R'} \xrightarrow{\text{par-r}} \frac{R \xrightarrow{\mu:\zeta} R'}{S \mid R \xrightarrow{\mu:\zeta} S \mid R'} = \frac{R \xrightarrow{\mu:\zeta} R' \quad \zeta \not \geqslant a}{(a)R \xrightarrow{\mu:\zeta} (a)R'} \qquad \frac{R_1 \equiv R \xrightarrow{\mu:\zeta} R' \equiv R_1'}{R_1 \xrightarrow{\mu:\zeta} R_1'} =$$

where in the (res) rule, $\zeta \not\supseteq a$ means that ζ is none of a, \bar{a} , a_* or \bar{a}_* . The last rule says that one can freely choose a representative in the structural congruence class before triggering a transition. It is used to push memories down threads using identities (1), (2) with their natural orientation.

Synchronization Rules. We end the description of transitions with the forward and backward synchronization rules.

Both kinds of synchronizations use a notion of *address instanciation* in monitored processes. Given a monitored process R and memories m_1, m_2 , a new process $R_{m_2 \otimes m_1}$ is obtained by replacing in R all memories of the form $\langle *, \alpha, Q \rangle \cdot m_1$ with $\langle m_2, \alpha, Q \rangle \cdot m_1$. This is used in forward synchronization to let the thread know the name of the other thread it synchronized with.

The complete definition is as follows:

$$\begin{array}{ll} ((a)R)_{m_2 @m_1} & \coloneqq (a)(R_{m_2 @m_1}) \\ (R \mid S)_{m_2 @m_1} & \coloneqq (R_{m_2 @m_1} \mid S_{m_2 @m_1}) \\ (\langle \triangleright P)_{m_2 @m_1} & \coloneqq \langle \triangleright \triangleright P \\ (\langle i \rangle \cdot m \triangleright P)_{m_2 @m_1} & \coloneqq \langle i \rangle \cdot m \triangleright P \\ (\langle m', \alpha, Q \rangle \cdot m \triangleright P)_{m_2 @m_1} & \coloneqq \langle m', \alpha, Q \rangle \cdot m \triangleright P \\ (\langle \ast, \alpha, Q \rangle \cdot m \triangleright P)_{m_2 @m_1} & \coloneqq \langle \ast, \alpha, Q \rangle \cdot m \triangleright P \\ (\langle \ast, \alpha, Q \rangle \cdot m \triangleright P)_{m_2 @m_1} & \coloneqq \langle m_2, \alpha, Q \rangle \cdot m \triangleright P \\ & \text{if } m \neq m_1 \\ (\langle \ast, \alpha, Q \rangle \cdot m \triangleright P)_{m_2 @m_1} & \coloneqq \langle m_2, \alpha, Q \rangle \cdot m_1 \triangleright P \\ \end{array}$$

TEAM LING

When *R* is a coherent process, there can be at most one memory of the appropriate form, and therefore $R_{m_2 \otimes m_1}$ and *R* differ at most at that particular location. With this definition in place, we can formulate neatly the synchronization rules:

$$\begin{array}{c|c} R \xrightarrow{m_{1}:\alpha} R' & S \xrightarrow{m_{2}:\bar{\alpha}} S' \\ \hline R \mid S \xrightarrow{m_{1},m_{2}:\tau} R'_{m_{2}} @m_{1} \mid S'_{m_{1}} @m_{2} \\ \hline R \xrightarrow{m_{1}:\alpha_{*}} R' & S \xrightarrow{m_{2}:\bar{\alpha}_{*}} S' \\ \hline R_{m_{2}} @m_{1} \mid S_{m_{1}} @m_{2} \xrightarrow{m_{1},m_{2}:\tau_{*}} R' \mid S' \end{array}$$

Backward Synchronization Discussed. As one can see in the definition, backward synchronization is also a communication. Once a thread T in R has used (syn), its memory is instanciated, and the resulting $T_{m_1 \otimes m_2}$ cannot backtrack if not with rule (syn_{*}). One doesn't roll back a synchronization all alone.

This "locking effect" can be illustrated with the following example:

$$\langle \rangle \triangleright (a.0 \mid \bar{a}.0) \equiv \langle 1 \rangle \triangleright a.0 \mid \langle 2 \rangle \triangleright \bar{a}.0 \xrightarrow{\langle 1 \rangle, \langle 2 \rangle: \tau} \langle \langle 2 \rangle, a, 0 \rangle \cdot \langle 1 \rangle \triangleright 0 \mid \langle \langle 1 \rangle, \bar{a}, 0 \rangle \cdot \langle 2 \rangle \triangleright 0$$

Both threads are locked together, and the only way backward for them is to synchronize again:

$$\begin{array}{c|c} \overset{\text{act}_{*}}{-} & \overbrace{\langle \ast, a, 0 \rangle \cdot \langle 1 \rangle \triangleright 0} \xrightarrow{\langle 1 \rangle : a_{*}} \langle 1 \rangle \triangleright a.0 & \overbrace{\langle \ast, \bar{a}, 0 \rangle \cdot \langle 2 \rangle \triangleright 0} \xrightarrow{\langle 2 \rangle : \bar{a}_{*}} \langle 1 \rangle \triangleright a.0 & \underset{\langle \langle 2 \rangle, a, 0 \rangle \cdot \langle 1 \rangle \triangleright 0}{\langle \langle 2 \rangle, a, 0 \rangle \cdot \langle 1 \rangle \triangleright 0} | \langle \langle 1 \rangle, \bar{a}, 0 \rangle \cdot \langle 2 \rangle \triangleright 0 \xrightarrow{\langle 1 \rangle, \langle 2 \rangle : \tau_{*}} \langle 1 \rangle \triangleright a.0 & \langle 2 \rangle \triangleright \bar{a}.0 & \underset{\langle \ast, a, 0 \rangle \cdot \langle 2 \rangle \triangleright \bar{a}.0}{\langle 2 \rangle, \bar{a}, 0 \rangle \cdot \langle 2 \rangle \triangleright 0} \xrightarrow{\langle 1 \rangle, \langle 2 \rangle : \tau_{*}} \langle 1 \rangle \triangleright a.0 & \langle 2 \rangle \triangleright \bar{a}.0 & \underset{\langle \ast, a, 0 \rangle \cdot \langle 2 \rangle \land \bar{a}.0}{\langle 2 \rangle, \bar{a}, 0 \rangle \cdot \langle 2 \rangle \land \bar{a}.0} \\ \end{array}$$

Relation to CCS. Write $R \to {}^*S$ if there exists a computation leading from R to S in RCCS, and likewise $P \to {}^*Q$ if there exists a computation from P to Q in CCS.

Lemma 4. If $P \to Q$ and $\varphi(R) = P$, then $R \to S$ for some S such that $Q = \varphi(S)$.

To see this, it is enough to forget about backward actions in the LTS above. Then it becomes the ordinary LTS for CCS with memories being useless additional notations. $\hfill \Box$

Thus any RCCS term R, such that $\varphi(R) = P$, can simulate P's behaviour in a purely forward manner. Of course, what one would like some form of converse guaranteeing the consistency of the backtracking mechanism. We will provide such a converse later in the paper (corollary 1).

Coherence. The transition system which we presented gives means of defining computation traces. Especially convenient is the fact that one doesn't have to work up to associativity and commutativity of the product and therefore gets a simple naming scheme for threads, where a thread can be uniquely identified by its memory. We have to verify that transitions don't disturb this naming scheme and preserve coherence.

Lemma 5. If $R \xrightarrow{\mu:\zeta} S$ and R is coherent, then so is S.

Basic transitions are only adding to, or chopping off the stack, a triplet of the form $\langle m, \alpha, Q \rangle$, and therefore clearly preserve coherence. Among the other rules, with the exception only the structural congruence rule (\equiv) and the synchronization rules (syn) and (syn_{*}) modify memories. The case of structural congruence was already dealt with in a previous lemma. Synchronizations only instantiate or de-instantiate one memory top at a time (because the process undergoing the transition is itself coherent), and this is again a transformation clearly preserving coherence.

As a consequence, any process obtained from some $\ell(P)$, where P is a CCS process, is coherent. Thereafter, we will assume all processes to be coherent.

3 Interlude

Let us have a short example before going to the causality section. Three convenient bits of notation will ease the reading. First, and although only the canonical notation gives a robust naming scheme, we use fork structures $\lambda(m)$ instead of full memories m as identifiers. Second, we use *n*-ary forks though this is not official notation. Last, when the choice left over in a choice action is 0, we just don't write it. That said we can run our example:

$$\begin{array}{l} \langle \rangle \triangleright \left(x.a.P \mid \bar{y}.\bar{x} \mid y.Q\right) & \equiv \\ \stackrel{(2),(3):\tau}{\longrightarrow} \langle 1 \rangle \triangleright x.a.P \mid \langle 2 \rangle \triangleright \bar{y}.\bar{x} \mid \langle 3 \rangle \triangleright y.Q \\ & \stackrel{(1),(2):\tau}{\longrightarrow} \langle 1 \rangle \triangleright x.a.P \mid \langle \langle 3 \rangle, \bar{y} \rangle \langle 2 \rangle \triangleright \bar{x} \mid \langle \langle 2 \rangle, y \rangle \langle 3 \rangle \triangleright Q \\ & \stackrel{(1),(2):\tau}{\longrightarrow} \langle \langle 2 \rangle, x \rangle \langle 1 \rangle \triangleright a.P \mid \langle \langle 1 \rangle, \bar{x} \rangle \langle \langle 3 \rangle, \bar{y} \rangle \langle 2 \rangle \triangleright 0 \mid \langle \langle 2 \rangle, y \rangle \langle 3 \rangle \triangleright Q \\ & \stackrel{(1):\bar{a}}{\longrightarrow} \langle \langle *, a \rangle \langle \langle 2 \rangle, x \rangle \langle 1 \rangle \triangleright P \mid \langle \langle 1 \rangle, \bar{x} \rangle \langle \langle 3 \rangle, \bar{y} \rangle \langle 2 \rangle \triangleright 0 \mid \langle \langle 2 \rangle, y \rangle \langle 3 \rangle \triangleright Q \\ \end{array}$$

One sees how the repeated synchronizations create a causal bottleneck: the synch on y caused the synch on x, which in turn caused the a action. Therefore, this sequence of three events is completely sequential and cannot be rearranged in any other order. Nothing is concurrent in this computation. That much is obvious. Less obvious is the fact that one doesn't need to go through the whole computation to realize this. It can be read off the stacks directly in the final process. The rightmost thread wants to backtrack on y with $\langle 2 \rangle$ (identifying the middle thread), the middle thread wants to backtrack on \bar{x} with $\langle 1 \rangle$ (identifying the leftmost thread), while the leftmost thread wants to backtrack on a all alone.

The point of the next section is to prove that this property holds in general. Backtracking an event is possible when and only when a causally equivalent trace would have brought this event as the last one. In our example there is no other equivalent trace, and indeed no other action than a can be backtracked.

This example illustrates another point made later. Suppose a is declared to be unbacktrackable, or irreversible, then the process now behaves as $P \mid Q$. The last section will extend RCCS and integrate such irreversible actions.

4 Causality

4.1 Transitions and Traces

We need first to go through a few standard definitions.

Recall that a transition t is given by a triplet $R \xrightarrow{\mu:\zeta} S$ with R, S monitored processes, μ an identifier (that is either a memory or a pair of memories) and ζ a directed action. One says then that R is the *source* and S the *target* of t and that S and R are its *ends*. Transitions will be ranged over in this section with t, t', ... and similar symbols. Two transitions are said to be *coinitial* if they have the same source, *cofinal* if they have the same target, *composable* if the source of the second is the target of the first. A transition is said to be *forward* or *backward* depending on whether its associated action μ is forward or backward.

Sequences of pairwise composable transitions will be called *computation traces* or simply traces, and will be ranged over by r, s, etc. All notions just defined for transitions extend readily to traces. In particular a trace will be said forward if all transitions it is composed of are forward. The empty trace with source R is denoted by ϵ_R and when r and s are composable, their composition is written r; s. A derivable transition is one that can be derived using the LTS of the second section. We have had examples of these already. From now on we will assume all transitions and traces to be derivable and with coherent ends (equivalently coherent sources) since these are the ones of interest.

If t is such a transition with identifier $\mu = m_1, m_2$, then it must have been obtained by a synchronization, and hence m_1 and m_2 have to be distinct, since one is assuming all processes to be coherent. So identifiers can be considered as sets (with either one or two elements) and be handled with set-theoretic operations. Transitions involving a one element memory will be termed *unary*, others will be termed *binary*.

Another noteworthy point is that a derivable transition is essentially derivable in a unique way. The only freedom is in the application of the \equiv rule and apart from the additive congruence, it only involves pushing memories past restrictions and products.

4.2 Causal Equivalence

Lemma 6 (Loop). For any forward transition $t : R \xrightarrow{\mu:\alpha} S$, there exists a backward transition $t_* : S \xrightarrow{\mu:\alpha_*} R$ and conversely.

Given s a forward trace, one can then obtain s_* a backward trace, by applying repeatedly the lemma and reversing all transitions in s.

Definition 7. Let $t_1 = R \xrightarrow{\mu_1:\zeta_1} S_1$ and $t_2 = R \xrightarrow{\mu_2:\zeta_2} S_2$ be two coinitial transitions, t_1 and t_2 are said to be concurrent if $\mu_1 \cap \mu_2 = \emptyset$.

Lemma 8 (Square). Let $t_1 = R \xrightarrow{\mu_1:\zeta_1} S_1$ and $t_2 = R \xrightarrow{\mu_2:\zeta_2} S_2$ be concurrent transitions, there exists two cofinal transitions $t_2/t_1 = S_1 \xrightarrow{\mu_2:\zeta_2} T$ and $t_1/t_2 = S_2 \xrightarrow{\mu_1:\zeta_1} T$.

Definition 9. Keeping with the notation of the square and loop lemmas above, one defines the causal equivalence, written ~, as the least equivalence relation between traces closed under composition and such that:

$$t_1; t_2/t_1 \sim t_2; t_1/t_2 \tag{3}$$

$$t; t_* \sim \epsilon_R \tag{4}$$

$$t_*; t \sim \epsilon_R \tag{5}$$

As said earlier, this is the analogue in CCS of the Berry-Lévy notion of equivalence of computation traces "by permutation" [2]. The "square" part of the definition was already considered by Boudol and Castellani [9] and they were well aware of the connection to Lévy's theory.

4.3 RCCS as a Lévy Labelling

Theorem 1. Let s_1 and s_2 be coinitial, then $s_1 \sim s_2$ iff s_1 and s_2 are cofinal.

By construction if $s_1 \sim s_2$, then s_1 and s_2 must be coinitial and cofinal, so the only if part is trivial. The if part is not. We first need a lemma.

Lemma 10. Let s be a trace, there exists r, r' both forward such that $s \sim r_*; r'$.

We prove this by lexicographic induction on the length of s, and the distance to the beginning of s of the earliest pair of transitions in s contradicting the property. If there is no such contradicting pair, then we are done. If there is one, say $t'; t_*$, and $\mu(t') \cap \mu(t_*) = \emptyset$, then t' and t_* can be swapped by virtue of the square lemma, resulting in a later earliest contradicting pair, and by induction the result follows since swapping keeps the length constant.

Suppose now $\mu(t') \cap \mu(t_*) \neq \emptyset$, and suppose further that t' is unary and write m for $\mu(t')$. By definition of a unary transition t' is pushing some triplet $\langle *, \alpha, Q \rangle$ on top of the stack at m, a triplet which t_* is then popping. This forces t_* to be unary as well, since there is no partner it may synch backwards with. But then t = t' and one can apply the loop lemma, hence the total length decreases and again by induction the result follows.

Suppose finally t' is binary, then t_* has to be as well, because of the locking effect discussed above, and again t = t' and the loop lemma applies and so does the inductive hypothesis.

Intuitively, the lemma says that, up to causal equivalence, one can always reach for the maximum freedom of choice, going backward, and only then go forward. Even more intuitively, one could picture such traces as parabolas, the process first draws potential energy from its memories and only then moves onward. So let us call such traces *parabolic*.³

³ The converse decomposition as $r; r'_*$ would not work. Indeed, a backward transition can create some new forward possibility. Consider for instance $\langle *, a, b.P \rangle > 0$ that can go to $\langle *, b, a.0 \rangle > P$ in a backward step followed by a forward one. There is no way these two steps can be swapped or be cancelling each other as in the argument above.

Our lemma already has an interesting corollary, namely that backtracking is consistent with respect to CCS, in the weaker sense that RCCS will not generate traces reaching processes the projections of which are unreachable in CCS.

Corollary 1. $P \to Q$ if and only if for some R, $\ell(P) \to R$ and $\varphi(R) = Q$.

The only if part is easy, basically saying that forward CCS computations can be simulated by RCCS ones, a fact that was recorded as an earlier lemma. The if part follows from the lemma above. Indeed, suppose s is an RCCS trace with source $\ell(P)$ and target R, then the lemma says $s \sim r_*; r'$ for some well chosen forward traces r and r', but then surely r must be empty since its source is $\ell(P)$ a process that has an empty memory and therefore is incapable of backtrack. So that there is forward trace r' equivalent to s. Since forward computations coincide in both systems, P reduces to $\varphi(R)^4$.

Lemma 11. Let s_1 , s_2 be coinitial and cofinal traces and s_2 be forward, then there exists a forward trace s'_1 , shorter than s_1 and such that $s'_1 \sim s_1$.

We prove this last lemma by induction on the length of s_1 . If s_1 is forward we are done. Else by lemma 10 we may suppose s_1 is parabolic. Let $t_*; t'$ be the only two successive transitions in s_1 with opposite directions and call μ the identifiers of t_* . Whatever t_* takes off the memories in μ has to be put back later down s_1 by some forward transition. Else, because s_2 is forward the difference will stay visible. Call t the earliest such transition in s_1 . For the same reason, that transition t has to be the exact inverse of t_* . One can then bubble up t to meet with t_* . Indeed, if a transition is conflicting with t on its way to t_* , by construction it must be some forward transition t'' the application of which results in a memory in μ , which is impossible since no backward transitions happened since t_* . So by applying repeatedly the square lemma, and then applying (5), one obtains a shorter equivalent s'_1 and conclude by induction.

Proof of Theorem. With our lemmas 10 and 11 in place, we can handle the theorem.

Let then s_1 , s_2 be two traces with same source and target. We prove these are causally equivalent, using a lexicographic induction on a pair consisting of the sum of the lengths of s_1 and s_2 , and the depth of the earliest disagreement between them. By lemma 10, we may suppose these are parabolic. Call t_1 and t_2 the earliest transitions were they disagree. There are three main cases in the argument depending on whether these are forward or backward.

1. If t_1 is forward and t_2 is backward, one has $s_2 = r_*; t_2; u$ and $s_1 = r_*; t_1; v$ for some r, u, v. Lemma 11 applies to $t_1; v$ which is forward, and $t_2; u$ which is parabolic, so $t_2; u$ has a shorter forward equivalent, and so s_2 has a shorter equivalent and the result follows by induction.

2. If t_1 , t_2 are both forward, they can't possibly be conflicting.

⁴ The fact that $\ell(P)$ has an empty memory is essential in the statement. We may consider again the same example $\langle *, a, P \rangle \triangleright 0$ which can move, while its projection $\varphi(\langle *, a, P \rangle \triangleright 0) = 0$ can't.

Suppose they take different additive choices in a same thread, say $m \triangleright \alpha . P + \alpha' . P' + Q$, respectively pushing on the memory $\langle *, \alpha, \alpha' . P' + Q \rangle$ and $\langle *, \alpha', \alpha . P + Q \rangle$. Since by assumption t_1 and t_2 are not the same transition, P and Q have to be different (not even additively congruent that is), and this difference would stay visible onward contradicting s_1 and s_2 being cofinal.

So whenever they work on a same thread they have to make the same additive choices. In particular, this rules out the case where they are both unary, and also the case where they are both binary and working on the same two threads. Let us check the other cases. If one of the transitions is unary and the other is binary, their actions can't coincide on their common memory, since one will push a triplet of the form $\langle *, a, Q \rangle$ and the other will push one of the form $\langle m, a, Q' \rangle$ for some $m \neq *$. If both are binary and only intersect on one memory, then again the triplets are distinct since the other partner identifiers, say m_1 and m_2 are different, because these identifiers come from different threads and all processes are assumed coherent. In any case, there would be a visible and permanent difference.

So we know they are indeed concurrent. Call μ_2 the identifier of t_2 . Since t_2 is forward and s_2 is parabolic, whatever t_2 pushes to μ_2 is staying there onward in s_2 . Hence, there must be a transition at μ_2 in s_1 as well, call it t'_2 , so that $s_1 = s; t_1; u; t'_2; v$. That transition has to be of the same arity as t_2 , and with $\mu(t'_2) = \mu_2$, else again the difference stays visible onward. Since s_1 is parabolic as well, all transitions in u, standing in between t_1 and t'_2 are forward and one can apply repeatedly the square lemma and bubble up t'_2 , to get s'_1 a trace equivalent to s_1 and of the form $s'_1 = s; t_1; t'_2; u; v$. A last application of the same square lemma yields an equivalent s''_1 with a later earliest divergence with s_2 and same length so that one can call on the induction hypothesis.

3. Suppose finally both t_1 and t_2 are backward. They can't possibly be conflicting either, because of the locking effect, so they must be concurrent. Either actions undone by t_2 are redone later in s_2 , in which case one argues as in lemma 11 and shortens s_2 into an equivalent s'_2 , or they are not redone and therefore these same actions must be undone also in s_1 , else there would be a visible difference between their visible targets, and one argues as in 2 above.

5 Irreversible Actions

Having finally reached for the theoretical property we wanted, we now turn to a more practical issue, namely that of integrating unbacktrackable or irreversible actions in RCCS.

These will be written \underline{a} , $\underline{\bar{a}}$, etc. The transition system is adapted by adding rules for the new actions:

$$\frac{R \xrightarrow{m_1:\kappa} R' \quad S \xrightarrow{m_2:\kappa} S'}{R \mid S \xrightarrow{m_1,m_2:\tau_0} R' \mid S'} \text{ since }$$

with κ ranging over irreversible actions. Since an irreversible action will never be backtracked, there is no need to remember anything, and we use instead of the

usual triplet, a placeholder $\langle o \rangle$. For the same reason, there is no rule inverse to (commit), and no longer a need to instanciate R' and S' in the synchronization rule. One sees that the syntax hardly changes, and that when all actions are taken to be irreversible, it essentially becomes CCS. What was proved in the last section remains valid.

We have seen in Section 3 that all actions that caused the action a became de *facto* unbacktrackable, even if they were themselves reversible. It seems therefore interesting to understand how irreversible actions can "domino-effect" in a process and this is what we do now.

5.1 Propagating Irreversible Actions

Let \mathcal{M}_R stand for the set of occurrences of memories in R. Say a memory is *locked* in R if no trace starting from R can modify it, that is to say, whenever $R \to^* S$, there is an $m' \in \mathcal{M}_S$ such that $m \leq m'$ for the prefix ordering. Let \mathcal{L}_R be the subset of \mathcal{M}_R formed of these locked memories.

Lemma 12. \mathcal{L}_R satisfies:

 $l. \ \langle \circ \rangle \cdot m \in \mathcal{M}_R \Rightarrow \langle \circ \rangle \cdot m \in \mathcal{L}_R$

2. $m \in \mathcal{L}_R, m' \leq m \Rightarrow m' \in \mathcal{L}_R$

- 3. $\langle m, \alpha, P \rangle \cdot m' \in \mathcal{L}_R \Rightarrow \langle m', \bar{\alpha}, Q \rangle \cdot m \in \mathcal{L}_R$
- $4. \quad \langle i \rangle \cdot m \in \mathcal{L}_R \Rightarrow \langle j \rangle \cdot m \in \mathcal{L}_R$

Points 1 and 2 are obvious, point 3 is saying that in order to backtrack a synch, one needs to do a synch, a fact which we called the locking effect, and point 4 is saying that to undo some action in $\langle i \rangle \cdot m$, one has first to fuse back with one's former sibling in the fork.

Coherence alone is not strong enough for these closure properties to capture all locked memories. For instance, m is locked in $\langle 1 \rangle \cdot m \triangleright P$ simply because the fork sibling is missing. However it does for CCS reachable processes.

5.2 An Example

To illustrate how the lemma works in practice to simplify processes, let us consider the usual encoding of internal sum in CCS:

$$R = \langle \rangle \triangleright (x)(\bar{x}.0|x.\underline{a}.P|x.\underline{b}.Q),$$

with x chosen *reversible* and a, b irreversible. With the usual notational simplifications, and representing locked memories with heavy angles (_):

$$\begin{array}{l} R \to^{\star} (x) \big(\langle \langle 2 \rangle, \bar{x} \rangle \langle 1 \rangle \triangleright 0 \mid \langle \circ \rangle \langle \langle 1 \rangle, x \rangle \langle 2 \rangle \triangleright P \mid \langle 3 \rangle \triangleright x. \underline{b}. Q \big) =: R_{a} \\ R \to^{\star} (x) \big(\langle \langle 3 \rangle, \bar{x} \rangle \langle 1 \rangle \triangleright 0 \mid \langle 2 \rangle \triangleright x. \underline{a}. P \mid \langle \circ \rangle \langle \langle 1 \rangle, x \rangle \langle 3 \rangle \triangleright Q \big) =: R_{b} \end{array}$$

Using clauses 2, 3, 4, one can tag further memories as locked:

$$\begin{split} R_{a} &= (x)(\langle \langle 2 \rangle, \bar{x} \rangle \langle 1 \rangle \triangleright 0 \mid \langle \circ \rangle \langle \langle 1 \rangle, x \rangle \langle 2 \rangle \triangleright P \mid \langle 3 \rangle \triangleright x. \underline{b}. Q) \\ R_{b} &= (x)(\langle \langle 3 \rangle, \bar{x} \rangle \langle 1 \rangle \triangleright 0 \mid \langle 2 \rangle \triangleright x. \underline{a}. P \mid \langle \circ \rangle \langle \langle 1 \rangle, x \rangle \langle 3 \rangle \triangleright Q) \end{split}$$

TEAM LING

so that in the end all memories are locked in $R_{\underline{a}}$ and $R_{\underline{b}}$, and R has been turned into a weak external sum !⁵

6 Conclusion

We have proposed an enrichment of CCS with memories which processes can use to backtrack. Memories are themselves distributed and the syntax stays close to the original concept of CCS. On the theoretical side, we have proved that backtracking is done in exact accordance with the true concurrency concept of causal equivalence. We also have shown how to integrate irreversible actions, and have given a procedure collecting obsolete memories.

There are many directions in which this study can be extended. First, one should rephrase this construction in terms of Winskel's event structures for CCS [17]. RCCS seems to provide an internal language for reversible event structure and this should be made a precise statement. Second, one should seriously study the notion of bisimulation that is generated by the LTS and study process composition in RCCS, perhaps using notions of history-preserving bisimulations [18]. Third, preliminary investigations show that the whole strategy developed here can be imported in π -calculus. There are more things to be remembered and the amount of symbol-pushing needed for a comparable theory gets daunting. So, beginning within the simpler framework of CCS might have been good start. But π is a lot more expressive and this is a strong incentive to develop a notion of reversible π -calculus. Besides, one could think of bootstrapping the system and encode reversible π into π .

Finally, the example of the internal sum given in the last section is strongly suggesting that transactional mechanisms can be understood in terms of RCCS. One starts with a rough encoding of the external sum, which is the simplest possible transaction in some sense. And since RCCS provides a foolproof dead-lock escape mechanism on reversible actions, carefully choosing them results in a correct encoding. We feel that one contribution of the current paper is to lay down the foundations to explore this matter further.

References

- 1. Robin Milner. *Communication and Concurrency*. International Series on Computer Science. Prentice Hall, 1989.
- 2. Jean-Jacques Lévy. Réductions optimales en λ-calcul. PhD, 1978.
- 3. Gérard Berry and Jean-Jacques Lévy. Minimal and optimal computation of recursive programs. *JACM*, 26:148–175, 1979.
- Aviv Regev, William Silverman, and Ehud Shapiro. Representation and simulation of biochemical processes using the π-calculus process algebra. In R. B. Altman, A. K. Dunker, L. Hunter, and T. E. Klein, editors, *Pacific Symposium on Biocomputing*, volume 6, pages 459–470, Singapore, 2001. World Scientific Press.

⁵ By weak external sum, we mean that R has infinite traces where it constantly hesitates between its irreversible actions a and b.

- 5. Corrado Priami, Aviv Regev, Ehud Shapiro, and William Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 2001.
- 6. Vincent Danos and Jean Krivine. Formal molecular biology done in CCS. In *Proceedings of BIO-CONCUR'03, Marseille, France, volume ? of Electronic Notes in Theoretical Computer Science.* Elsevier, 2003. To appear.
- 7. K.V.S. Prasad. Combinators and bisimulation proofs for restartable systems. PhD, 1987.
- Jan A. Bergstra, Alban Ponse, and Jos van Wamel. Process algebra with backtracking. In *REX School Symposium*, pages 46–91, 1993.
- Gérard Boudol and Ilaria Castellani. Permutation of transitions: An event structure semantics for CCS and SCCS. *Lecture Notes in Computer Science: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, 354:411– 427, 1989.
- 10. Pierpaolo Degano, Rocco De Nicola, and Ugo Montanari. A partial ordering semantics for CCS. *Theoretical Computer Science*, 75:223–262, 1990.
- 11. Gérard Boudol, Ilaria Castellani, Matthew Hennesy, and Astrid Kiehn. Observing localities. In *Proceedings MFCS'91*, volume 114, pages 31–61, 1991.
- 12. Pierpaolo Degano and Corrado Priami. Proved trees. In *Automata, Languages and Programming*, volume 623 of *LNCS*, pages 629–640. Springer Verlag, 1992.
- 13. Gérard Boudol, Ilaria Castellani, Matthew Hennesy, and Astrid Kiehn. A theory of processes with localities. *Formal Aspect of Computing*, 1992.
- 14. Ilaria Castellani. Observing distribution in processes: Static and dynamic localities. *International Journal of Foundations of Computer Science*, 6(4):353–393, 1995.
- 15. Pierpaolo Degano and Corrado Priami. Non interleaving semantics for mobile processes. In *Automata, Languages and Programming,* volume 944 of *LNCS*, pages 660–667. Springer Verlag, 1995.
- 16. Michele Boreale and Davide Sangiorgi. A fully abstract semantics for causality in the π -calculus. Acta Informatica, 35, 1998.
- 17. Glynn Winskel. Event structure semantics for CCS and related languages. In *Proceedings of 9th ICALP*, volume 140 of *LNCS*, pages 561–576. Springer, 1982.
- 18. Marek A. Bednarczyk. Hereditary history preserving bisimulations or what is the power of the future perfect in program logics. ICS PAS Report, April 1991.

Parameterised Boolean Equation Systems (Extended Abstract)

Jan Friso Groote¹ and Tim Willemse^{1,2}

 ¹ Department of Mathematics and Computer Science, Eindhoven University of Technology
 P.O. Box 513, 5600 MB Eindhoven, The Netherlands J.F.Groote@tue.nl

² Faculty of Science, Mathematics and Computing Science, University of Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands

Abstract. Boolean equation system are a useful tool for verifying formulas from modal mu-calculus on transition systems (see [9] for an excellent treatment). We are interested in an extension of boolean equation systems with data. This allows to formulate and prove a substantially wider range of properties on much larger and even infinite state systems. In previous works [4,6] it has been outlined how to transform a modal formula and a process, both containing data, to a so-called parameterised boolean equation system, or equation system for short. In this article we focus on techniques to solve such equation systems.

We introduce a new equivalence between equation systems, because existing equivalences are not compositional. We present techniques similar to Gauß elimination as outlined in [9] that allow to solve each equation system provided a single equation can be solved. We give several techniques for solving single equations, such as approximation (known), patterns (new) and invariants (new). Finally, we provide several small but illustrative examples of verifications of modal mu-calculus formulas on concrete processes to show the use of the techniques.

1 Introduction

Boolean Equation Systems (BESs) [9-11] are systems of the form

$$(\sigma_1 X_1 = f_1) \dots (\sigma_N X_N = f_N),$$

where σ_i is either a least fixpoint symbol μ or a greatest fixpoint symbol ν and f_i is a propositional formula. These systems can be seen as generalisations of nested and alternating fixpoint expressions, interpreted over a Boolean lattice.

BESs have been studied in detail by Vergauwen and Lewi [11], and Mader [9, 10] in the context of model checking modal μ -calculus formulae. In [10], Mader shows that the model checking problem can be solved by solving BESs.

Furthermore, she provides a complete proof system for solving BESs by means of algebraic manipulations.

Parameterised Boolean Equation Systems (PBESs) (also known as *First-Order* Boolean Equation Systems) [4,6,12] are sequences of equations of the form

$$\sigma X(d_1:D_1,\ldots,d_n:D_n)=\varphi,$$

where σ is either a least or a greatest fixpoint symbol, d_i is a data variable of sort D_i and φ is a predicate formula. The sort $D_1 \times \cdots \times D_n$ is referred to as the *parameter-space* of a parameterised boolean equation.

PBESs form an extension of plain BESs. Groote and Mateescu [4] introduced these PBESs as an intermediate formalism for model checking processes with (arbitrary) data. Extending on the results of Mader [9,10], they showed that their model checking problem could be translated to the problem of solving PBESs. In [4], they provided four proof rules for approximating the solutions of single parameterised equations: two for the least fixpoint and two for the greatest fixpoint. In [6,12] we showed that PBESs can be solved automatically by an algorithm that combines the essentials of Gauß-elimination [9,10] and approximation (see e.g. [3]).

While this algorithm worked well for certain equation systems, it did not work for others as it for instance required transfinite approximations of fixpoint expressions. The emphasis on automation set a scene where possible remedies for such situations where hard to find.

Inspired by this latter observation, we take a different approach altogether in this paper, and focus on algebraic techniques to solve PBESs by hand. While this may seem a step back to some, being able to solve PBESs by hand provides a better understanding of the techniques that are involved. We intentionally proved many properties about systems manually, some of which can be found in the second part of this paper, with as primary goal to extend the range of effective techniques to solve parameterised boolean equation systems. Although it is not the focus of this paper, we expect that these techniques will also have a positive impact on the mechanised and automatic verification of modal formulas on processes in a setting with data.

The approach we describe in this paper is similar in spirit to the algebraic approach for solving BESs, taken by Mader [10]. We separate the problems of solving PBESs as a whole, and parameterised boolean equations in isolation. Central to our approach are the notions of a *system equivalence* and *system ordering* that allow us to reason compositionally about PBESs. While in [10], also a system equivalence is introduced for BESs, it turns out that this equivalence is not compositional. We illustrate this fact by a simple example in section 3.

Based on our new notion of system equivalence, we present an overall and complete technique, allowing to solve all PBESs using syntactic manipulations only, provided the means to solve a single parameterised boolean equation in isolation are available.

We consequently investigate various techniques for solving a single parameterised boolean equation. These include a theorem allowing logical reasoning
using predicate calculus and results on approximation from [4] in terms of the new system equivalence.

Some of the parameterised boolean equation systems that we encountered were not easily solved using known techniques. But we noticed that many of these had a very similar pattern. For some of the most common patterns we could give a standard solution. We, however, believe that we have only scratched this topic on the surface. We expect a situation comparable to solving differential equations, where identifying and solving differential equations of a particular form has become a field of its own. There have been a number of typical patterns that we have not been able to solve and that deserve a separate investigation.

While invariants are an effective tool in diverse areas, esp. program analysis [2], they have not yet been connected to BESs and PBESs. So, we set out to find their counterpart in parameterised boolean equations. Our notion of an invariant in equation systems plays a very helpful role in many of the examples in section 5 and so we believe that it will become a similarly effective tool as invariants are elsewhere.

The structure of this paper is as follows. Section 2 introduces the terminology used throughout this paper, together with a short overview of PBESs, their semantics and several smaller results. In section 3 an equivalence for PBESs is introduced and compared against the equivalence for BESs that can be found in the literature. Section 4 then focuses on techniques for solving PBESs globally and parameterised boolean equations in isolation. As an illustration of these techniques, we apply these to several smaller examples in section 5. A full version of this paper appeared as [7]. It contains more results, examples and all the proofs.

Acknowledgements. We thank Marc Voorhoeve, Joost-Pieter Katoen and Kees van Hee for discussions regarding this paper, and anonymous referees for their useful comments.

2 Definition of a Parameterised Boolean Equation System

We are interested in solving sequences of fixpoint equations where the equations have the form

 $\mu X(d_1:D_1,\ldots,d_n:D_n)=\varphi$

where μ indicates a minimal fixpoint, or

$$\nu X(d_1:D_1,\ldots,d_n:D_n)=\varphi$$

where ν indicates that this is a maximal fixpoint equation.

Each equation has a predicate variable X (from a set \mathcal{X} of variables) at its left hand side that depends on zero or more data variables d_1, \ldots, d_n of sorts D_1, \ldots, D_n . For simplicity and without loss of generality, we restrict ourselves to a single variable at the left hand side in all our theoretical considerations. We treat data in an abstract way. So, we assume that there are non empty data

sorts, generally written using letters D, E, F, that include the sort \mathbb{B} of booleans containing \perp and \top , representing *false* and *true*, respectively. We have a set \mathcal{D} of data variables, with typical elements d, d_1, \ldots , and we assume that there is some data language that is sufficiently rich to denote all relevant data terms. For a closed term e, we assume an interpretation function [e] that maps e to the data element it represents. For open terms we use a *data environment* ε that maps each variable from \mathcal{D} to a data value of the right sort. The interpretation of an open term e of sort \mathbb{B} , denoted as $[e] \varepsilon$ is given by $[\varepsilon(e)]$ where ε is extended to terms in the standard way.

The right hand side of each equation is a *predicate formula* containing data terms, boolean connectives, quantifiers over (possibly infinite) data domains and data and predicate variables. Predicate formulae φ are defined by the following grammar:

$$\varphi ::= b \mid X(e) \mid \varphi \land \varphi \mid \varphi \lor \varphi \mid \forall d : D.\varphi \mid \exists d : D.\varphi \mid \top \mid \bot$$

where b is a data term of sort \mathbb{B} , X is a predicate variable, d is a data variable of sort D and e is a data term. Note that negation does not occur in predicate formulae, except possibly as an operator in data terms.

In the sequel it turns out to be necessary to lift predicate formulas to functions from data to formulas. We use conventional typed lambda calculus to denote such functions. E.g. $\lambda d: D.\varphi$ denotes a function from elements from data sort D to predicates. Sometimes, the lambda is omitted if that leads to a more compact notation. For instance $\lambda d: D.X(d)$ is generally simply written as X.

Predicate formulae are interpreted in a context of a data environment ε and a *predicate environment* $\eta: \mathcal{X} \rightarrow (D \rightarrow \mathbb{B})$. The semantics of predicate formulae is defined below. For an arbitrary environment θ (be it a data environment or predicate environment), we write $\theta[v/d]$ for the environment θ in which the variable *d* has been assigned the value *v*. For a predicate formula φ , a predicate environment η and a data environment ε , we write $\varphi(\eta\varepsilon)$, denoting the formula φ in which all free predicate variables *X* have received the value $\eta(X)$ and all free data variables *d* have received the value $\varepsilon(d)$. Environments are applied to functions, where bound variables are respected.

Definition 1. Let ε be a data environment and $\eta: \mathcal{X} \to (D \to \mathbb{B})$ be a predicate environment. The interpretation $[\![\varphi]\!]\eta\varepsilon$ maps a predicate formula φ to "true" or "false" and is inductively defined as follows:

$$\begin{split} [\top] \eta \varepsilon & \stackrel{\text{def}}{=} true \\ [\bot] \eta \varepsilon & \stackrel{\text{def}}{=} false \\ [b] \eta \varepsilon & \stackrel{\text{def}}{=} [b] \varepsilon \\ [X(e)] \eta \varepsilon & \stackrel{\text{def}}{=} \eta(X)([e] \varepsilon) \\ [\varphi_1 \land \varphi_2] \eta \varepsilon & \stackrel{\text{def}}{=} [\varphi_1] \eta \varepsilon \text{ and } [\varphi_2] \eta \varepsilon \\ [\varphi_1 \lor \varphi_2] \eta \varepsilon & \stackrel{\text{def}}{=} [\varphi_1] \eta \varepsilon \text{ or } [\varphi_2] \eta \varepsilon \end{split}$$

$$\begin{bmatrix} \forall d: D.\varphi \end{bmatrix} \eta \varepsilon \stackrel{\text{def}}{=} \begin{cases} \text{true, if for all } v:D \text{ it holds that } \llbracket \varphi \rrbracket \eta(\varepsilon[v/d]) \\ \text{false, otherwise} \\ \\ \exists d: D.\varphi \rrbracket \eta \varepsilon \stackrel{\text{def}}{=} \begin{cases} \text{true, if there exists a } v:D \text{ such that } \llbracket \varphi \rrbracket \eta(\varepsilon[v/d]) \\ \text{false, otherwise} \end{cases}$$

Consider for an arbitrary data sort D, all (total) functions $f:D \to \mathbb{B}$. The set of all such functions is denoted $[D \to \mathbb{B}]$. The ordering \sqsubseteq on $[D \to \mathbb{B}]$ is defined as $f \sqsubseteq g$ iff for all d:D, we have f(d) implies g(d). The set $([D \to \mathbb{B}], \sqsubseteq)$ is a complete lattice. For a subset A of $[D \to \mathbb{B}]$, we write (ΛA) for the *infimum* of the set A and $(\bigvee A)$ for the *supremum* of the set A.

We denote the set of all predicate environments by $[\mathcal{X} \to (D \to \mathbb{B})]$. The ordering \leq on $[\mathcal{X} \to (D \to \mathbb{B})]$ is defined as $\eta \leq \eta'$ iff for all $X \in \mathcal{X}$, we have $\eta(X) \sqsubseteq \eta'(X)$. The set $([\mathcal{X} \to (D \to \mathbb{B})], \leq)$ is also a complete lattice.

Definition 2. A parameterised boolean equation system is inductively defined as follows: the empty parameterised boolean equation system is denoted ϵ , and for a parameterised boolean equation system \mathcal{E} , also $(\sigma X(d:D) = \varphi)\mathcal{E}$ is a parameterised boolean equation system where $\sigma \in \{\mu, \nu\}$ is a fixpoint symbol and φ a predicate formula.

In the remainder of this article, we abbreviate parameterised boolean equation system with *equation system* if no confusion can arise. The set of *binding* predicate variables in an equation system \mathcal{E} , denoted by $bnd(\mathcal{E})$, is defined as $\operatorname{bnd}(\epsilon) \stackrel{\text{def}}{=} \emptyset$ and $\operatorname{bnd}((\sigma X(d:D) = \varphi)\mathcal{E}) \stackrel{\text{def}}{=} \operatorname{bnd}(\mathcal{E}) \cup \{X\}$, i.e. a binding variable is a variable that occurs at the left-hand side of an equation. An equation system \mathcal{E} is said to be *well-formed* iff all binding predicate variables of \mathcal{E} are unique. Thus, $(\nu X = \top)(\mu X = \bot)$ is not a well-formed equation system. We only consider well-formed equation systems in this paper. We say an equation system \mathcal{E} is *closed* whenever all predicate variables occurring at the right-hand side of the equations in \mathcal{E} (collected in the set $occ(\mathcal{E})$) are binding variables, i.e. $occ(\mathcal{E}) \subseteq bnd(\mathcal{E})$; if an equation system \mathcal{E} is not closed, we say \mathcal{E} is open. We say an equation $\sigma X(d:D) = \varphi$ is *solved* if φ contains no predicate variables. Likewise, an equation system \mathcal{E} is *solved* iff all its constituting equations are solved. We say that a parameterised boolean equation system is *solved in X* if the predicate variable X does not occur in any right hand side. The solution of an equation system is defined in the context of a predicate environment η and a data environment ε :

Definition 3. The solution of an equation system \mathcal{E} in the context of a predicate environment η and a data environment ε is inductively defined as follows (cf. definition 3.3 of [10]):

$$\begin{split} [\epsilon] \eta \varepsilon & \stackrel{def}{=} \eta \\ [(\sigma X(d:D) = \varphi) \mathcal{E}] \eta \varepsilon \stackrel{def}{=} [\mathcal{E}](\eta [\sigma X(d:D).\varphi([\mathcal{E}]\eta \varepsilon)/X]) \end{split}$$

where $\sigma X(d:D).\varphi([\mathcal{E}]\eta\varepsilon)$ is defined as

$$\mu X(d:D).\varphi([\mathcal{E}]\eta\varepsilon) \stackrel{\text{def}}{=} \bigwedge \{\psi: D \to \mathbb{B} \mid \lambda v: D.\llbracket \varphi \rrbracket ([\mathcal{E}]\eta[\psi/X]\varepsilon[v/d])\varepsilon[v/d] \sqsubseteq \psi \}$$
$$\nu X(d:D).\varphi([\mathcal{E}]\eta\varepsilon) \stackrel{\text{def}}{=} \bigvee \{\psi: D \to \mathbb{B} \mid \psi \sqsubseteq \lambda v: D.\llbracket \varphi \rrbracket ([\mathcal{E}]\eta[\psi/X]\varepsilon[v/d])\varepsilon[v/d] \}$$

In the remainder of this paper, we only consider *data-closed* equation systems, i.e. equation systems in which all data variables that occur on the right hand side of an equation are bound in the left hand side of this equation. This means that we can use the empty data environment for all our considerations without affecting the solution of the equation system. In general, we do not explicitly write down the empty data environment.

As an illustration, consider the equation system $(\nu X=Y)(\mu Y = X)$. For a given predicate environment η , its solutions are $\eta[\top/X][\top/Y]$. Note that the solution for $(\mu Y=X)(\nu X=Y)$ is $\eta[\perp/X][\perp/Y]$. This illustrates that the sequence in which the equations occur is of importance.

3 Equivalences for Equation Systems

Boolean equation systems (BESs) have been studied in great detail [10]. BESs are instances of our parameterised boolean equation systems, i.e. the proposition variables in a BES do not carry data parameters. We introduce two notions of equivalence. The first equivalence is based on the equivalence between BESs, and can be found in [10]. We argue that this equivalence is not suitable and introduce an equivalence that is slightly finer.

Definition 4. Let $\mathcal{E}, \mathcal{E}'$ be equation systems. We write $\mathcal{E} \ll \mathcal{E}'$ iff for all predicate environments η it holds that $[\mathcal{E}]\eta \leq [\mathcal{E}']\eta$. We write $\mathcal{E} \sim \mathcal{E}'$ iff both $\mathcal{E} \ll \mathcal{E}'$ and $\mathcal{E}' \ll \mathcal{E}$. The relation \ll is referred to as the standard (equation) system ordering, whereas the relation \sim is referred to as the standard (equation) system equivalence.

It follows immediately from the definition of \ll and \sim that \ll is a partial ordering and \sim is an equivalence relation. However, the standard system equivalence \sim does not allow for compositional reasoning.

To illustrate this, consider the two open BESs $\mu X = Y$ and $\nu X = Y$. It is easy to see that $\mu X = Y \sim \nu X = Y$, since both have the same solutions for all predicate environments. However, this does not imply that the two BESs are equivalent in all contexts, since the predicate variable Y can interfere. For example, if we add the equation $\nu Y = X$ to the two BESs, the resulting BESs are different, i.e. we have $(\mu X = Y)(\nu Y = X) \nsim (\nu X = Y)(\nu Y = X)$, since the solution to the first BES is $X = Y = \bot$, whereas the solution to the second BES is $X = Y = \top$.

To mend this situation, we redefine the standard system equivalence and the standard system ordering. Throughout this paper we use this new notion and not the one from [10].

Definition 5. Let $\mathcal{E}, \mathcal{E}'$ be equation systems. We write $\mathcal{E} \Rightarrow \mathcal{E}'$ iff for all predicate environments η and all equation systems \mathcal{F} with $bnd(\mathcal{F}) \cap (bnd(\mathcal{E}) \cup bnd(\mathcal{E}')) = \emptyset$, it holds that $[\mathcal{EF}]\eta \leq [\mathcal{E}'\mathcal{F}]\eta$. We write $\mathcal{E} \equiv \mathcal{E}'$ iff both $\mathcal{E} \Rightarrow \mathcal{E}'$ and $\mathcal{E}' \Rightarrow \mathcal{E}$. The relation \Rightarrow is referred to as the (equation) system ordering, whereas the relation \equiv is referred to as (equation) system equivalence.

The relation \Rightarrow is reflexive, anti-symmetric and transitive and \equiv is an equivalence relation. Furthermore, these relations are compositional, as the following theorem states.

Theorem 1. Let $\mathcal{E}, \mathcal{E}', \mathcal{F}$ be equation systems for which $bnd(\mathcal{F}) \cap (bnd(\mathcal{E}) \cup bnd(\mathcal{E}')) = \emptyset$. Then

1. $\mathcal{E} \Rightarrow \mathcal{E}' \Rightarrow \mathcal{F}\mathcal{E} \Rightarrow \mathcal{F}\mathcal{E}',$ 2. $\mathcal{E} \Rightarrow \mathcal{E}' \Rightarrow \mathcal{E}\mathcal{F} \Rightarrow \mathcal{E}'\mathcal{F}.$

Proof. The second property follows immediately from the definition of \Rightarrow . Thus, we concentrate on the first property. We use induction on the length of \mathcal{F} .

- 1. Assume \mathcal{F} is the empty equation system. We must show that $\mathcal{E} \Rightarrow \mathcal{E}'$, but this holds by assumption,
- 2. Let η be a predicate environment. Assume \mathcal{F} is of the form $(\sigma X(d:D)=\varphi)\mathcal{F}'$. By definition, $[(\sigma X(d:D)=\varphi)\mathcal{F}'\mathcal{E}]\eta$ equals $[\mathcal{F}'\mathcal{E}]\eta[\sigma X(d:D).\varphi([\mathcal{F}'\mathcal{E}]\eta)/X]$. Using the induction hypothesis and the monotonicity of equation systems over environments, this is at most

$$[\mathcal{F}'\mathcal{E}]\eta[\sigma X(d:D).\varphi([\mathcal{F}'\mathcal{E}']\eta)/X].$$

By another application of the induction hypothesis, we find that this is at most $[\mathcal{F}'\mathcal{E}']\eta[\sigma X(d:D).\varphi([\mathcal{F}'\mathcal{E}']\eta)/X]$. This equals $[(\sigma X(d:D)=\varphi)\mathcal{F}'\mathcal{E}']\eta$, by definition. Thus

$$(\sigma X(d:D) = \varphi) \mathcal{F}' \mathcal{E} \Longrightarrow (\sigma X(d:D) = \varphi) \mathcal{F}' \mathcal{E}',$$

which concludes the proof.

Corollary 1. For all equation systems $\mathcal{E}, \mathcal{E}', \mathcal{F}$, we have

1. $\mathcal{E} \equiv \mathcal{E}' \Rightarrow \mathcal{F}\mathcal{E} \equiv \mathcal{F}\mathcal{E}'$, and 2. $\mathcal{E} \equiv \mathcal{E}' \Rightarrow \mathcal{E}\mathcal{F} \equiv \mathcal{E}'\mathcal{F}$

provided $\operatorname{bnd}(\mathcal{F}) \cap (\operatorname{bnd}(\mathcal{E}) \cup \operatorname{bnd}(\mathcal{E}')) = \emptyset$.

The standard system equivalence and ordering are very much related to the system equivalence and ordering, as defined in definition 5. We find that for closed equation systems the two notions coincide.

Lemma 1. Let \mathcal{E} and \mathcal{E}' be closed equation systems. Then $\mathcal{E} \Rightarrow \mathcal{E}'$ iff $\mathcal{E} \ll \mathcal{E}'$.

4 Proof Techniques for Equation Systems

4.1 Global Techniques for Solving Equation Systems

The focus in this section is on algebraic techniques for solving equation systems as a whole. The first lemma also appeared (in the setting of boolean equation

systems) in [10] as lemma 6.3 using a slightly different phrasing. It allows to substitute the right hand side of an equation for the left hand side in all the equations preceding it. In [10], this step formed an essential part of the so-called *Gauß elimination* procedure to solve boolean equation systems.

Lemma 2 (Substitution). Let \mathcal{E} be an equation system for which $X, Y \notin bnd(\mathcal{E})$, then:

$$(\sigma X(d:D) = \varphi)\mathcal{E}(\sigma'Y(e:E) = \psi) \equiv (\sigma X(d:D) = \varphi[\psi/Y])\mathcal{E}(\sigma'Y(e:E) = \psi)$$

The sequence in which equations in a parameterised boolean equation system occur is important. It is only allowed to change this order under very particular circumstances. The following lemma in this section deals with reordering of equations.

Lemma 3 (Migration). Let $\sigma X(d:D) = \varphi$ be a solved equation, i.e. $occ(\varphi) = \emptyset$, and \mathcal{E} an equation system, such that $X \notin bnd(\mathcal{E})$, then:

$$(\sigma X(d:D) = \varphi)\mathcal{E} \equiv \mathcal{E}(\sigma X(d:D) = \varphi)$$

The following theorem states that we have the means to solve an equation system if we can solve a single equation.

Theorem 2. Assume we can derive for all equations $(\sigma X(d:D)=\varphi) \equiv (\sigma X(d:D)=\psi)$, such that X does not occur in ψ . Then all closed equation systems can be rewritten to solved equation systems using the rules of migration and substitution.

The full version of this paper [7] contains some additional results that apply globally to an equation system. These allow to change the order of equations and to transfer results in terms of system ordering to system equivalence.

4.2 Local Techniques for Solving Equation Systems

In theorem 2 it has been shown that we can solve a parameterised boolean equation system, if we can solve each equation of the form $\sigma X(d:D) = \varphi$ in X, i.e. if we can find an equivalent equation in which X does not occur in the right hand side. In this section, we focus on techniques to find such equations.

We do not strive for completeness in any formal sense here. Our focus in this paper is to yield a set of rules that allows effective manual verification, and we have shown efficacy by applying our rules to numerous examples some of which occur in section 5. General incompleteness results indicate that completeness can only be achieved under particular circumstances. For instance, it is possible to prove completeness using infinitary logics (see e.g. [8]). But such means are unwieldy for practical purposes and generally only satisfy a general desire for completeness results. Completeness can also be achieved for restricted data types. This is useful as such exercises can reveal new verification rules and techniques. Albeit interesting, we do not treat such questions in this paper and postpone these to further investigations in the field. A self evident way of solving a single equation is by applying the standard rules of predicate calculus. In order to use these, we first define logical implication for our setting.

Definition 6. Let φ, φ' be arbitrary predicate formulae. We write $\varphi \rightarrow \varphi'$, representing logical implication which is defined as $[\![\varphi]\!]\eta\varepsilon$ implies $[\![\varphi']\!]\eta\varepsilon$ for all data environments ε and predicate environments η . We write $\varphi \leftrightarrow \varphi'$ as a shorthand for $\varphi \rightarrow \varphi'$ and $\varphi' \rightarrow \varphi$.

Note that in this definition we used a data environment, which is only important if free data variables occur in formulae. In line with the rest of this paper, we omit the data environment elsewhere.

Lemma 4. Let φ and ψ be predicate formulae such that $\varphi \rightarrow \psi$. Then $(\sigma X(d:D) = \varphi) \Rightarrow (\sigma X(d:D) = \psi)$.

A straightforward but often laborious method for solving an equation $\sigma X(d:D) = \varphi$ in X is by means of an iterative approximation of the fixpoint solution of X, which is possible as we are dealing with a monotonic operators over a poset. One starts with an initial solution S_0 for X being either $\lambda d:D. \perp$ (for $\sigma = \mu$) or $\lambda d:D. \top$ (for $\sigma = \nu$). Then the approximate solutions of the form $\lambda d:D.S_{n+1} = \varphi[S_n/X]$ are calculated repeatedly. A *stable approximant* is an approximant that is logically equivalent to its next approximation. A stable approximant is in fact the fixpoint solution to the equation.

Definition 7. Let φ, ψ be predicate formulae and X a predicate variable. We inductively define $\psi[\varphi/X]^k$, where k is of sort \mathbb{N} .

1. $\psi[\varphi/X]^0 \stackrel{def}{=} \varphi$, and 2. $\psi[\varphi/X]^{k+1} \stackrel{def}{=} \psi[(\psi[\varphi/X]^k)/X].$

Thus, $\psi[\varphi/X]^k$ represents the result of recursively substituting φ for X in ψ . Note that for any k:N, and all predicate formulae ψ, φ , the expression $\psi[\varphi/X]^k$ is a predicate formula. Below we state that $\varphi[\perp/X]^k$ and $\varphi[\top/X]^k$ are approximations of the solution of an equation and that a stable approximant is *the* solution to an equation.

Lemma 5. Let φ be a predicate formula and $k:\mathbb{N}$ be an arbitrary natural number. *Then*

 $\begin{array}{l} 1. \ (\mu X(d:D) = \varphi[\bot/X]^k) \Rrightarrow (\mu X(d:D) = \varphi). \\ 2. \ (\nu X(d:D) = \varphi) \Rrightarrow (\nu X(d:D) = \varphi[\top/X]^k). \\ 3. \ If \varphi[\bot/X]^k \leftrightarrow \varphi[\bot/X]^{k+1} \ then \ (\mu X(d:D) = \varphi[\bot/X]^k) \equiv (\mu X(d:D) = \varphi). \\ 4. \ If \varphi[\top/X]^k \leftrightarrow \varphi[\top/X]^{k+1} \ then \ (\nu X(d:D) = \varphi) \equiv (\nu X(d:D) = \varphi[\top/X]^k). \end{array}$

Invariants characterise 'the reachable parameter space' of a parameterised boolean equation. As in the verification of programs they can be used to prove properties that only hold within the reachable state space. Within parameterised boolean equation systems they can be used to simplify equations with a particular parameter instantiation.

A formal definition of an invariant is given below. In our setting the definition looks uncommon, but still expresses what is ordinarily understood as an invariant. Note that our invariants only have the transfer property, and do not involve an initial state.

Definition 8. Let $\sigma X(d:D) = \varphi$ be an equation and let $I:D \rightarrow \mathbb{B}$ be a predicate formula in which no predicate variable occurs. Then, I is an invariant of X iff

 $(I \land \varphi) \leftrightarrow (I \land \varphi[(\lambda e: D.I[e/d] \land X(e))/X])$

The theorem below says that if χ is a solution for the equation $\sigma X(d:D) = \varphi$ under invariant *I* (condition 1) and *X* is used in an equation $\sigma' Y(e:E) = \psi$ in a situation where *I* implies *X* (condition 2), then we may substitute solution χ for *X* in ψ .

Theorem 3. Let $\sigma X(d:D) = \varphi$ and $\sigma' Y(e:E) = \psi$ be equations and let $I:D \to \mathbb{B}$ be an invariant of X. Let \mathcal{E} be a parameterised boolean equation system such that $\{X,Y\} \not\subseteq \operatorname{bnd}(\mathcal{E})$. If for some predicate formula χ such that $X \notin \operatorname{occ}(\chi)$

1.
$$(\sigma X(d:D) = \varphi \land I(d)) \equiv (\sigma X(d:D) = \chi)$$
 and
2. $(\sigma' Y(e:E) = \psi) \equiv (\sigma' Y(e:E) = \psi[\lambda d:D.I(d) \land X(d)/X]).$

then

$$(\sigma'Y(e:E) = \psi)\mathcal{E}(\sigma X(d:D) = \varphi) \equiv (\sigma'Y(e:E) = \psi[\lambda d:D.\chi/X])\mathcal{E}(\sigma X(d:D) = \varphi)$$

We encountered several typical equation systems for which none of the techniques for finding the solution we discussed so far apply. For instance, iterative approximation is not always applicable, as the following example shows.

Example 1. Consider the following greatest fixpoint equation: $\nu X(i:\mathbb{N}) = i \leq N \wedge X(i+1)$, where N is some arbitrary natural number. By approximating, we obtain infinitely many approximants, without ever reaching the solution. Obviously, the solution to this equation should be $\forall j:\mathbb{N}.i+j \leq N$, which can be further reduced to \bot .

In order to be able to solve such an equation effectively, we need to resort to a different method altogether. We provide generic *patterns* of equations and solutions to these. Equations, such as the one from the above example, can then be recognised to be of a certain form, and be solved by looking them up. Note that identifying 'patterns' is very common in mathematics, for instance when solving differential equations.

The first pattern is obtained by generalising the equation in the example given above. Note that the solutions for the minimal and maximal fixpoint equations are dual. Let $f:D \rightarrow D$ be an arbitrary, total function. We assume the existence of a function $f:\mathbb{N} \times D \rightarrow D$, written as $f^n(d)$, with the property that $f^0(d) = d$ and $f^{n+1}(d) = f(f^n(d))$.

Theorem 4. Let $\sigma X(d:D) = \varphi(d) \land (\psi(d) \lor X(f(d)))$ be an equation, where $f:D \rightarrow D$ is an arbitrary total function and X does not occur in φ and ψ .

- 1. The solution to X for $\sigma = \nu$ is $\forall j:\mathbb{N}$. $((\exists i:\mathbb{N}. i < j \land \psi(f^i(d))) \lor \varphi(f^j(d))),$ 2. The solution to X for $\sigma = \mu$ is:
 - $\exists i: \mathbb{N}. \ \psi(f^i(d)) \land \forall j: \mathbb{N}. \ (j \leq i \to \varphi(f^j(d))).$

When more than one occurrence of X occurs in the right hand side of the pattern in theorem 4 we have a straightforward generalisation for which we can find a solution in a similar vein.

In this case we assume that functions $f_i: D \to D$ for i < N for some given N are given. We let $g: \mathbb{N} \to \{0, \dots, N-1\}$ be an arbitrary function. We assume the existence of functions f(g, j, d) with the property that f(g, 0, d) = d and $f(g, j + 1, d) = f_{g(j)}(f(g, j, d))$.

Theorem 5. Let $N:\mathbb{N}$ be some arbitrary natural number and let

$$\sigma X(d:D) = \varphi(d) \wedge \bigwedge_{i=0}^{N-1} (\psi_i(d) \vee X(f_i(d)))$$

be an equation, where $f_i: D \rightarrow D$ are arbitrary total functions and X does not occur in φ and ψ_i .

- 1. The solution to X for $\sigma = \nu$ is $\forall j: \mathbb{N} \cdot \forall g: \mathbb{N} \rightarrow \{0, \dots, N-1\}. ((\exists i: \mathbb{N} \cdot i < j \land \psi_{g(i)}(f(g, i, d))) \lor \varphi(f(g, j, d))),$
- 2. The solution to X for $\sigma = \mu$ is $\exists j:\mathbb{N}:\exists g:\mathbb{N}\to\{0,\ldots,N-1\}$. $(\langle \forall i:\mathbb{N}:i < j \to \neg \psi_{g(i)}(f(g,i,d))) \land \varphi(f(g,j,d)))$,

A pattern that we encountered but were not able to solve thus far is the following:

$$\sigma X(d:D) = \varphi(d) \land \forall e: E.(\psi(d,e) \lor X(f(d,e)))$$

for arbitrary data sort *E*. Actually, — and we pose this as a very interesting open question — it might be possible to device a method to solve all single fixed point equations of the form $\sigma X(d:D) = \varphi$ by replacing φ by a first order formula in which *X* does not occur. Using Gauß elimination, this would yield a complete method that allows to transform each parameterized boolean equation system to a first order formula.

5 Applications

In this section, we study three systems by proving the validity of certain modal formulas governing their behaviour. We translate the process descriptions and the formulas to parameterised boolean equation systems that are subsequently solved. For a detailed account on how these equations can be derived from a process and a formula, we refer to [4, 6, 12]. Although our examples do not use parallelism, the available techniques are perfectly suited for it. For the remainder of this paper, we assume the reader is familiar with the use of the specification language μ CRL (see e.g. [5]), and the use of the *first-order modal* μ -calculus with data [4, 6] to specify logical properties of systems.

5.1 Merging Infinite Streams

Combining several input streams into a single stream is a technique that is found frequently in streaming media applications. The way streams are combined depends on a particular application. Here, we study a small system that reads data from two (infinite) input streams, one-by-one, and produces a new output stream that is locally ascending, see figure 1. Our particular merge system is



Fig. 1. Combining Two Input Streams into a Single Output Stream

described by the four process equations below. The initial process is *Merge*. It reads data from stream i via action r_i , where $i \in \{1, 2\}$, and the output is produced via action s.

$$\begin{array}{l} Merge = \sum_{m:\mathbb{N}} & (r_1(m) \cdot Merge_1(m) + r_2(m) \cdot Merge_2(m)) \\ Merge_1(n:\mathbb{N}) = \sum_{m:\mathbb{N}} & r_2(m) \cdot Merge_3(n,m) \\ Merge_2(m:\mathbb{N}) = \sum_{n:\mathbb{N}} & r_1(n) \cdot Merge_3(n,m) \\ Merge_3(n,m:\mathbb{N}) = s(n) \cdot Merge_2(m) \triangleleft n \leq m \triangleright s(m) \cdot Merge_1(n) \end{array}$$

The process *Merge* reads an arbitrary natural number $m:\mathbb{N}$ via channel r_1 (expressed using the sum or choice operator Σ) and proceeds by executing process $Merge_1$. Or (expressed by +) it reads a value m via channel r_2 and proceeds with $Merge_2$. In the definition of $Merge_3$ the triangles $______$ represents the *then-if-else*, saying that if $n \leq m$, $s(n) \cdot Merge_2(m)$ is chosen, and otherwise $s(m) \cdot Merge_1(n)$ is executed.

Clearly, on ascending input streams, the merge system should produce an ascending output. This is expressed by the following formula where modalities such as $[r_1(l)]\varphi$ mean that whenever action $r_1(l)$ can be performed in a certain state, φ must hold in the next state:

$$egin{aligned} &(
u ilde{X}(in_1,in_2,out:\mathbb{N}).orall l:\mathbb{N}.\;([r_1(l)](l\geq in_1 o ilde{X}(l,in_2,out))\wedge\ &[r_2(l)](l\geq in_2 o ilde{X}(in_1,l,out))\wedge\ &[s(l)](l\geq out\wedge ilde{X}(in_1,in_2,l))))(0,0,0) \end{aligned}$$

TEAM LING

The process *Merge* must first be converted to linear form if we are to verify this property. This is fairly straightforwardly achieved by introducing an additional parameter σ :N. Process *Merge_i* is represented by $\sigma = i$, whereas $\sigma = 0$ represents process *Merge* itself. Combining the resulting linear process specification with the above formula according to the translation of [4, 6, 12] together with some simplifications, yields:

$$\begin{split} \nu X(\sigma,n,m,in_1,in_2,out:\mathbb{N}) = \\ & (\sigma=0 \rightarrow (\forall l:\mathbb{N}.\ l \geq in_1 \rightarrow X(1,l,m,l,in_2,out))) \land \\ & (\sigma=0 \rightarrow (\forall l:\mathbb{N}.\ l \geq in_2 \rightarrow X(2,n,l,in_1,l,out))) \land \\ & (\sigma=1 \rightarrow (\forall l:\mathbb{N}.\ l \geq in_2 \rightarrow X(3,n,l,in_1,l,out))) \land \\ & (\sigma=2 \rightarrow (\forall l:\mathbb{N}.\ l \geq in_1 \rightarrow X(3,l,m,l,in_2,out))) \land \\ & (\sigma=3 \land n \leq m) \rightarrow (n \geq out \land X(2,n,m,in_1,in_2,m)) \land \\ & (\sigma=3 \land m \leq n) \rightarrow (m \geq out \land X(1,n,m,in_1,in_2,m)) \end{split}$$

where the ascending input/output property holds if $X(\sigma, n, m, 0, 0, 0)$ holds.

A closer inspection of the equation reveals a striking similarity in the use of the variables n and in_1 , and, likewise, in the variables m and in_2 . This is in fact no coincidence. In the linear process, representing process *Merge*, the variables n and m register the last read values of stream 1 and stream 2, respectively. The variables in_1 and in_2 , appearing in the modal formula have a similar purpose. This redundancy is identified by the invariant $(n = in_1) \land (m = in_2)$. Furthermore, the variable *out* satisfies the invariant *out* $\leq \min(in_1, in_2)$. It is straightforward to verify that both properties are invariants in the sense of definition 8. Thus, rather than immediately solving this equation, it pays off to solve the equation with the invariant.

$$\begin{split} \nu X_I(\sigma,n,m,in_1,in_2,out:\mathbb{N}) &= \\ & (n{=}in_1 \wedge m{=}in_2 \wedge out {\leq} \min(in_1,in_2)) \wedge \\ & (\sigma=0 \rightarrow (\forall l{:}\mathbb{N}.\ l \geq in_1 \rightarrow X_I(1,l,m,l,in_2,out))) \wedge \\ & (\sigma=1 \rightarrow (\forall l{:}\mathbb{N}.\ l \geq in_2 \rightarrow X_I(2,n,l,in_1,l,out))) \wedge \\ & (\sigma=2 \rightarrow (\forall l{:}\mathbb{N}.\ l \geq in_1 \rightarrow X_I(3,l,m,l,in_2,out))) \wedge \\ & (\sigma=3 \wedge n \leq m) \rightarrow (n \geq out \wedge X_I(2,n,m,in_1,in_2,m)) \wedge \\ & (\sigma=3 \wedge m \leq n) \rightarrow (m \geq out \wedge X_I(1,n,m,in_1,in_2,m)) \end{split}$$

It is straightforward to approximate this equation, where X_i denotes the *i*'th approximation.

$$\begin{array}{l} X_0(\sigma,n,m,in_1,in_2,out) = \top, \\ X_1(\sigma,n,m,in_1,in_2,out) = n = in_1 \wedge m = in_2 \wedge out \leq \min(in_1,in_2). \end{array}$$

The approximation X_1 is stable and hence it is the solution for X_1 .

Now we cannot use this solution to construct a solution for $X(\sigma, n, m, 0, 0, 0)$, simply because it does not satisfy the invariant. However, if we consider $X(\sigma, 0, 0, 0, 0, 0, 0)$, then using theorem 3 we can use the solution for X_I as the solution for X. More concretely, $X(\sigma, 0, 0, 0, 0, 0)$ is always true. Approximating the fixpoint equation for X directly does not terminate as quickly and is awkward due to a universal quantifier that remains present in the approximations.

5.2 An Identity Tag Generator

Many applications depend on a mechanism that produces identity tags for objects. Illustrative examples of such tags are phone-numbers, but also IP-addresses and message-header tags in e-mails. In essence, the mechanism for producing identity tags is a process that writes an infinite stream of identities. We represent these identities by means of natural numbers, see figure 2. The process



Fig. 2. Identity tag generator

Generator is a generic process that generates identity tags according to some predefined function that is passed as a parameter to process Generator. The generator is initialised with the value i.

proc Generator($f:\mathbb{N}\to\mathbb{N}, i:\mathbb{N}$) = $s(i) \cdot Generator(f, f(i))$

Thus, by executing process *Generator*(*succ*,0), where *succ* is the successor function for natural numbers, we can generate the natural numbers. Most applications, using the generator, rely on the generator to produce unique tags. Thus, any two outputs of the system should be different. This is expressed by the following modal formula. It says that always in the future whenever a tag m is generated, every tag n generated later is not equal to m. The modality $[T]\varphi$ holds in a state if for each action that can be performed φ holds in the subsequent state.

$$u ilde{X}.([\top] ilde{X} \wedge orall m : \mathbb{N}.[s(m)]
u ilde{Y}.([\top] ilde{Y} \wedge orall n : \mathbb{N}.[s(n)] m
eq n))$$

An alternative but more complex formulation of this property would be to store all outputs in a set and check that each tag being generated does not occur in the set. The fact that this is not needed in the above modal formula is due to the greatest fixpoint operators which allow to state properties about all infinite runs of a system. Verifying this modal formula on process *Generator* allows us to find the conditions on the generator function that ensures all produced tags are unique. In order to do so, we need to solve the following equation system:

$$\begin{array}{ll} \nu X(f:\mathbb{N} \rightarrow \mathbb{N},i:\mathbb{N}) &= X(f,f(i)) \land \forall m:\mathbb{N}.(m=i) \rightarrow Y(f,f(i),m), \\ \nu Y(f:\mathbb{N} \rightarrow \mathbb{N},i,m:\mathbb{N}) &= Y(f,f(i),m) \land \forall n:\mathbb{N}.(n=i) \rightarrow m \neq n. \end{array}$$

TEAM LING

Obviously, all universal quantifiers can be removed in the equations above. Thus, we can rewrite this equation system to the following equivalent equation system.

$$\nu X(f:\mathbb{N} \to \mathbb{N}, i:\mathbb{N}) = X(f, f(i)) \land Y(f, f(i), i), \\ \nu Y(f:\mathbb{N} \to \mathbb{N}, i, m:\mathbb{N}) = Y(f, f(i), m) \land m \neq i.$$

These equations are both of the form of the pattern of theorem 4. Hence, the solution to Y is $\forall j:\mathbb{N}$. $f^{j}(i) \neq m$. The solution to X is $\forall j':\mathbb{N}.\forall j:\mathbb{N}.f^{j+j'+1}(i) \neq f^{j'}(i)$, which is logically equivalent to $\forall j:\mathbb{N}.\forall j':\mathbb{N}. j \neq j' \rightarrow f^{j}(i) \neq f^{j'}(i)$. This is exactly the requirement we expected, and it is nice to see that we can also systematically derive it.

5.3 A Lossy Channel

Consider a simple lossy channel that reads information from a stream, and tries to send it to the other side where a message is lost occasionally.

$$C_{\mathsf{T}} = \sum_{m:\mathbb{N}} r(m) \cdot C_{\perp}(m)$$

$$C_{\perp}(m:\mathbb{N}) = s(m) \cdot C_{\mathsf{T}} + l \cdot C_{\mathsf{T}}$$

We wish to verify that when data is not always lost, messages eventually get across. We formulate this using the following modal formula

$$\nu \tilde{X}.([\top]\tilde{X} \land (\mu \tilde{Y}.[\top]\tilde{Y} \lor \langle l \rangle \top \lor \exists m: \mathbb{N}.\langle s(m) \rangle \top))$$

We first translate the process to linear form:

$$C(b:\mathbb{B}, m:\mathbb{N}) = \sum_{k:\mathbb{N}} r(k) \cdot C(\bot, k) \triangleleft b \triangleright \delta$$
$$s(m) \cdot C(\top, m) \triangleleft \neg b \triangleright \delta$$
$$l \cdot C(\top, m) \triangleleft \neg b \triangleright \delta$$

The process C_{\top} is equal to $C(\top, m)$ for any $m:\mathbb{N}$ and $C_{\perp}(m)$ is equal to $C(\perp, m)$.

The equation system we obtain is the following:

$$\begin{split} \nu X(b:\mathbb{B},m:\mathbb{N}) &= (\forall k:\mathbb{N}.(b \to X(\bot,k)) \land (\neg b \to X(\top,m))) \land Y(b,m) \\ \mu Y(b:\mathbb{B},m:\mathbb{N}) &= \\ (\forall k:\mathbb{N}.(b \to Y(\bot,k)) \land (\neg b \to Y(\top,m))) \lor \neg b \lor \exists m':\mathbb{N}.\neg b \land m = m' \end{split}$$

Approximation quickly leads to a solution without involving m:

$$\begin{array}{l} Y_0(b,m) = \bot, \\ Y_1(b,m) = \neg b \land (b \lor \neg b) = \neg b, \\ Y_2(b,m) = (\neg b \to \neg b) \lor \neg b = \top, \\ X_0(b,m) = \top \end{array}$$

where $X_0(b,m) = \top$ is a stable solution. Thus, in whatever state the process *C* starts, messages always get across if not always lost.

A slightly more involved property, taken from [1, page 309], says that delivery via action s(m) is fairly treated if there are no paths where s(m) is enabled infinitely often, but occurs only finitely often:

$$\nu \tilde{X}.\mu \tilde{Y}.\nu \tilde{Z}.\forall m:\mathbb{N}.[s(m)]\tilde{X} \land (\exists m:\mathbb{N}.\langle s(m)\rangle\top \to \\ ([l]\tilde{Y} \land \forall m:\mathbb{N}.[r(m)]\tilde{Y})) \land [l]\tilde{Z} \land \forall m:\mathbb{N}.[r(m)]\tilde{Z}$$

This formula and process C are translated to the following equation system

$$\begin{split} \nu X(b;\mathbb{B},m;\mathbb{N}) &= Y(b,m) \\ \mu Y(b;\mathbb{B},m;\mathbb{N}) &= Z(b,m) \\ \nu Z(b;\mathbb{B},m;\mathbb{N}) &= (\neg b \to X(\top,m)) \land (\neg b \to ((\neg b \to Y(\top,m)) \land \\ & \forall k;\mathbb{N}.(b \to Y(\bot,k)))) \land ((\neg b \to Z(\top,m)) \land \forall k;\mathbb{N}.(b \to Z(\bot,k))) \\ &= (\neg b \to X(\top,m) \land Y(\top,m) \land Z(\top,m)) \land (b \to \forall k;\mathbb{N}.Z(\bot,k)) \end{split}$$

We approximate Z and find a stable solution in three steps:

$$\begin{split} Z_{0}(b:\mathbb{B}, m:\mathbb{N}) &= \top, \\ Z_{1}(b:\mathbb{B}, m:\mathbb{N}) &= \neg b \to X(\top, m) \wedge Y(\top, m), \\ Z_{2}(b:\mathbb{B}, m:\mathbb{N}) &= (\neg b \to X(\top, m) \wedge Y(\top, m)) \wedge (\forall k:\mathbb{N}.X(\top, k) \wedge Y(\top, k)) \\ &= \forall k:\mathbb{N}.X(\top, k) \wedge Y(\top, k). \end{split}$$

We substitute the solution for Z in the second equation obtaining the equation:

$$\mu Y(b:\mathbb{B}, m:\mathbb{N}) = \forall k:\mathbb{N}. X(\top, k) \land Y(\top, k).$$

Using one approximation step it is easily seen that the solution of this equation is $Y(b,m) = \bot$. So, substitution of this solution in the first equation yields $X(b,m) = \bot$. The property does not hold for our process.

References

- J. Bradfield and C. Stirling. Modal logics and mu-calculi: an introduction. In, J.A. Bergstra, A. Ponse and S.A. Smolka, Handbook of process algebra, pp. 293– 330, Elsevier, 2001.
- P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1981.
- E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *First IEEE Symposium on Logic in Computer Science*, *LICS'86*, pages 267–278. IEEE Computer Society Press, 1986.
- J.F. Groote and R. Mateescu. Verification of temporal properties of processes in a setting with data. In A.M. Haeberer, *AMAST'98*, volume 1548 of *LNCS*, pp. 74–90. Springer-Verlag, 1999.
- J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 17, pages 1151–1208. Elsevier (North-Holland), 2001.
- 6. J.F. Groote and T.A.C. Willemse. A checker for modal formulas for processes with data. Technical Report CSR 02-16, Eindhoven University of Technology, Department of Mathematics and Computer Science, 2002.

- 7. J.F. Groote and T.A.C. Willemse. Parameterised boolean equation systems. Computer Science Report 04/09, Department of Mathematics and Computer Science, Eindhoven University of Technology, 2004.
- 8. D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
- A. Mader. Modal μ-calculus, model checking and gauß elimination. In E. Brinksma, R.W. Cleaveland, K.G. Larsen, T. Margaria, and B. Steffen, *Tools and Algorithms* for Construction and Analysis of Systems, First International Workshop, TACAS '95, Aarhus, Denmark, volume 1019 of Lecture Notes in Computer Science, pages 72–88. Springer-Verlag, 1995.
- 10. A. Mader. Verification of Modal Properties Using Boolean Equation Systems. PhD thesis, Technical University of Munich, 1997.
- B. Vergauwen and J. Lewi. Efficient local correctness checking for single and alternating boolean equation systems. In S. Abiteboul and E. Shamir, editors, *Proceedings ICALP'94*, volume 820 of *Lecture Notes in Computer Science*, pages 302–315. Springer-Verlag, 1994.
- 12. T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing.* PhD thesis, Eindhoven University of Technology, February 2003.

An Extensional Spatial Logic for Mobile Processes

Daniel Hirschkoff

LIP - ENS Lyon, France

Abstract. Existing spatial logics for concurrency are intensional, in the sense that they induce an equivalence that coincides with structural congruence. In this work, we study a contextual spatial logic for the π -calculus, which lacks the spatial operators to observe emptyness, parallel composition and restriction, and only has composition adjunct and hiding. We show that the induced logical equivalence coincides with strong early bisimilarity. The proof of completeness involves the definition of non-trivial formulas, including characteristic formulas for restriction-free processes up to bisimilarity. This result allows us to isolate the extensional core of spatial logics, decomposing spatial logics into a part that counts (given by the intensional operators) and a part that observes (given by their adjuncts). We also study how enriching the core extensional spatial logic with intensional operators affects its separative power.

1 Introduction

Spatial logics extend classical logic with constructions to reason about the structure of the underlying model (when applied to concurrent systems, the models are processes). The additional connectives belong to two families. *Intensional operators* allow one to inspect the structure of the model. A formula $A_1 | A_2$ is satisfied whenever we can split the structure into two parts satisfying the corresponding subformula A_i , i = 1, 2. In presence of restriction in the underlying model, a structure P satisfies formula $n \otimes A$ if we can write P as $(\nu n) P'$ with P' satisfying A. Finally, formula 0 is only satisfied by the empty structure. Connectives | and \otimes come with adjunct operators, called guarantee (\triangleright) and hiding (\otimes) respectively, that allow one to extend the structure being observed. In this sense, these can be called *contextual operators*. P satisfies $A_1 \triangleright A_2$ whenever the spatial composition (using |) of P with any structure satisfying A_1 satisfies A_2 , and P satisfies $A \otimes n$ if $(\nu n) P$ satisfies A.

Previous studies have demonstrated that in existing spatial logics, the intensional character prevails. In the static case, where spatial logics are used to reason about semi-structured data [CG01a], or about memory along the execution of a program that manipulates pointers [Rey02], the guarantee operator is eliminable, in the sense that every formula involving \triangleright can be replaced by an equivalent formula that does not make use of \triangleright [Loz03, Loz04, DGG04]. In spatial logics for concurrency [CG00, CC01], that also include a temporal modality,

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 325-339, 2004.

[©] Springer-Verlag Berlin Heidelberg 2004

this is not the case. However, the equivalence on processes induced by the logic coincides with structural congruence, a very fine grained relation on processes — much finer in particular than behavioural equivalence [San01, HLS02, CL04]. This situation is in contrast with standard modal logics for concurrency like the Hennessy-Milner (HM for short) logic [MPW93], for which logical equivalence is known to coincide with bisimilarity.

Technically, the ability for spatial logic to capture structural congruence on processes is based on two aspects of its expressiveness. The first aspect is the ability to *count*, i.e., to express arithmetical properties about the number of substructures exhibited by a given system. The second aspect is the definability of modalities à la Hennessy-Milner within the logic, i.e., one is able to capture parts of the behaviour of processes. This has been shown in [San01, HLS02], and further studied in [HLS03], using a logic with a restricted set of operators, and applying it to both the Ambient calculus and the π -calculus (modality formulas are also derived in [CL04]). In [HLS03], in particular, the derivability of modality formulas for the π -calculus and for Mobile Ambients heavily relies on the use of intensional operators, in conjunction with guarantee: | and 0 are used to isolate some kind of elementary components of interaction (called 'threads'), while the revelation operator makes it possible to test the free names of a process, in order to deduce behavioural properties.

In this work, we renounce to the intensional connectives, and study the resulting contextual spatial logic, called \mathcal{L} . \mathcal{L} only has spatial composition adjunct (\triangleright), revelation adjunct (\otimes), a simple temporal modality (\diamond), and an operator \mathcal{U} for fresh name quantification. We apply \mathcal{L} to reason about the π -calculus, and we show *extensionality* of the logic, in the sense that \mathcal{L} induces the same separative power as strong early bisimilarity (and thus as Hennessy-Milner logic). This result suggests that the two families of operators in spatial logics serve different purposes: while intensional operators allow one to count (as illustrated by the study in [DLM04], where it is shown that a particular static spatial logic, in which \triangleright is eliminable, characterises Presburger arithmetic), we show that contextual operators are enough to bring extensionality.

To establish our main result, we exploit the characterisation of strong bisimilarity (written ~) in terms of barbed equivalence (written \simeq). The elementary observations available in \mathcal{L} are indeed reminiscent of the definition of \simeq . However, technically, we still need to define a way to perform *instantaneous* observations (to detect barbs) in \mathcal{L} , which is a priori not obvious given the definition of the logic. We are only able to define formulas for barbs when imposing a bound on the size of processes, but this is enough for our purposes. Another aspect of the expressive power we need in order to capture \simeq is the ability to let two processes 'pass the same tests'. This is achieved by defining characteristic formulas for restriction-free processes up to ~. These formulas exploit the constructions for barbs, and are relatively concise thanks to some specific properties of bisimilarity on the calculus without restriction. As hinted above, due to the absence of intensional operators, our constructions depart from the formulas for modalities defined in related works [San01, HLS02, HLS03, CL04]. While we use \simeq in order to show that logical equivalence for \mathcal{L} coincides with ~, the argument does not follow the classical proof that \simeq is included in ~, and we instead use the ideas we just sketched. We briefly study also \mathcal{L}^{\ddagger} , an adaptation of \mathcal{L} that is closer to the observations given in \simeq (detecting barbs is primitive in \mathcal{L}^{\ddagger}). We show that \mathcal{L}^{\ddagger} is also an extensional logic.

Having isolated a core extensional spatial logic, we may wonder what lies between \mathcal{L} and full spatial logics for concurrency. To address this question, we establish some results about the expressive and separative power we obtain when enriching \mathcal{L} with (some) intensional operators. These results suggest that from the point of view of separability, the most powerful intensional operator is \mathbb{B} .

Outline. We introduce the calculus and the logic we study in Section 2. Formulas for (some of the) π -calculus modalities and to characterise bisimilarity classes of restriction-free processes are presented in Section 3. In Section 4, we exploit these constructions to prove that \mathcal{L} is extensional. Section 5 is devoted to the discussion of variants and enrichments of \mathcal{L} , and we conclude in Section 6.

2 Preliminaries

2.1 The π -Calculus

The finite synchronous π -calculus is introduced using an infinite set of names, ranged over using $a, b, \ldots, m, n, \ldots$ Processes, ranged over using P, Q, R, \ldots , are defined by the following syntax:

 $P \quad ::= \quad \mathbf{0} \quad \left| \begin{array}{c} P_1 | P_2 \end{array} \right| \quad (\boldsymbol{\nu} n) P \quad \left| \begin{array}{c} m(n) \cdot P \end{array} \right| \quad \overline{m} \langle n \rangle \cdot P \, .$

Trailing occurrences of **0** will often be omitted. Name *n* is bound in an inputprefixed term m(n).P, and in a restricted term $(\nu n) P$. A name that is not bound is free, and fn(*P*) will denote the set of free names of *P*. We write $P_{\{n \leftarrow m\}}$ for the process resulting from the capture-avoiding replacement of *n* with *m* in *P*.

Actions of the labelled transition system, ranged over with μ , are defined by the following syntax (notice the presence of free input):

 $\mu \quad ::= \quad mn \quad | \quad \overline{m} \langle n \rangle \quad | \quad \overline{m} (n) \quad | \quad \tau \, .$

Given an action μ , we define its names $(n(\mu))$, free names $(fn(\mu))$ and bound names $(bn(\mu))$ as usual. Figure 1 presents the transition rules that define the operational semantics of the π -calculus (symmetrical versions of rules involving parallel composition are omitted). We write $P \xrightarrow{\overline{m}(n)} P'$ whenever $P \xrightarrow{\overline{m}(n)} P'$ or $P \xrightarrow{\overline{m}(n)} P'$.

Structural congruence, \equiv , is the least equivalence relation that is a congruence and that satisfies the rules of Figure 2. Given a (possibly empty) sequence of names $\tilde{n} = n_1, \ldots, n_k$, $(\nu \tilde{n}) P$ will stand for $(\nu n_1) \ldots (\nu n_k) P$. We will also implicitly reason up to permutation of consecutive restrictions, thus treating \tilde{n} as a set of names.

$$\overline{m}\langle n \rangle . P \xrightarrow{\overline{m}\langle n \rangle} P \qquad m(n) . P \xrightarrow{\underline{m}a} P_{\{n \leftarrow a\}} \qquad \frac{P \xrightarrow{\mu} P'}{P|Q \xrightarrow{\mu} P'|Q} \operatorname{bn}(\mu) \cap \operatorname{fn}(Q) = \emptyset$$
$$\frac{P \xrightarrow{\mu} P'}{(\nu n) P \xrightarrow{\mu} (\nu n) P'} n \notin \operatorname{n}(\mu) \qquad \frac{P \xrightarrow{\overline{m}\langle n \rangle}}{(\nu n) P \xrightarrow{\overline{m}\langle n \rangle} P'} m \neq n$$
$$\frac{P \xrightarrow{\overline{m}\langle n \rangle}}{P|Q \xrightarrow{\tau} P'|Q'} \qquad \frac{P \xrightarrow{\overline{m}\langle n \rangle}}{P|Q \xrightarrow{\tau} (\nu n) (P'|Q')} n \notin \operatorname{fn}(Q)$$

Fig. 1. Early operational semantics

 $P|\mathbf{0} \equiv P \qquad P|Q \equiv Q|P \qquad P|(Q|R) \equiv (P|Q)|R \qquad (\nu n) \mathbf{0} \equiv \mathbf{0}$ $(\boldsymbol{\nu}n)(\boldsymbol{\nu}m)P \equiv (\boldsymbol{\nu}m)(\boldsymbol{\nu}n)P$ $P|(\boldsymbol{\nu}n) Q \equiv (\boldsymbol{\nu}n) (P|Q) \text{ if } n \notin \mathrm{fn}(P)$



The *public* π -calculus consists in the set of restriction-free processes. We shall also call P a public process whenever $P \equiv Q$ for some Q in the public π -calculus. Given a process P, we write size(P) for the number of prefixes of P. By definition, if $P \equiv Q$, then size(P) = size(Q). A process P is an *atom* if size(P) = 1.

We define some basic observations, usually called barbs, as follows: we write $P \downarrow_n$ (resp. $P \downarrow_{\overline{n}}$) whenever $P \equiv (\nu \widetilde{m}) (n(a) \cdot P_1 | P_2)$ (resp. $P \equiv (\nu \widetilde{m}) (\overline{n} \langle a \rangle \cdot P_1 | P_2)$) for some n, a, \tilde{m}, P_1 and P_2 such that $n \notin \tilde{m}$.

We shall write relation composition using juxtaposition, and the negation of a relation \asymp will be written $\not \asymp$. We do not give the usual definition of reduction, and instead equivalently (see [SW01]) set $P \longrightarrow Q \stackrel{\text{def}}{=} P \stackrel{\tau}{\longrightarrow} \equiv Q$.

2.2 **Behavioural Relations**

Definition 1 (Behavioural Equivalences).

- Strong bisimilarity, ~, is the greatest symmetrical relation such that whenever $P \sim Q$ and $P \xrightarrow{\mu} P'$, there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \sim Q'$.
- Strong barbed bisimilarity, $\dot{\simeq}$, is the greatest symmetrical relation such that whenever $P \simeq Q$:

 - (i) For any n, $(P \downarrow_n iff Q \downarrow_n)$ and $(P \downarrow_{\overline{n}} iff Q \downarrow_{\overline{n}})$. (ii) For any P' s.t. $P \longrightarrow P'$, there exists Q' s.t. $Q \longrightarrow Q'$ and $P' \simeq Q'$.
- P and Q are strong barbed equivalent, written $P \simeq Q$, iff for any process $R, P|R \simeq Q|R.$

In the sequel, we shall often omit the word 'strong' when mentioning these equivalences. The labelled transition system-based and reduction-based presentations for behavioural equivalence coincide, as expressed by the following result.

Theorem 1 ([SW01]). $P \sim Q$ iff $P \simeq Q$.

We shall need the following results about behavioural equivalence.

Proposition 1. Define \sim_s like \sim except that for actions μ of the form mn, when comparing two processes P and Q, we only consider names n belonging to the (finite) set $\operatorname{fn}(P) \cup \operatorname{fn}(Q) \cup \{d\}$, where $d \notin \operatorname{fn}(P) \cup \operatorname{fn}(Q)$. Then $\sim = \sim_s$.

Lemma 1. $\sim \subseteq \dot{\simeq}$, and $\sim \dot{\simeq} \sim \subseteq \sim$.

Lemma 2. Given a process P, we have the following:

- 1. There exist names \tilde{n} and a public process P_0 such that $P \sim (\nu \tilde{n}) P_0$.
- 2. If size(P) < 2 * k for some integer k, then P cannot perform a sequence of reductions of length equal to k.

Proof. The first result follows from the two laws $P|(\nu n)Q \sim (\nu n)(P|Q)$ (when $n \notin \operatorname{fn}(P)$) and $\alpha . (\nu n)P \sim (\nu n) \alpha . P$ where α is a prefix of the form m(m') or $\overline{m}\langle m' \rangle$ with $n \notin \{m, m'\}$.

Note that the results of this lemma hold because we work in a finite calculus.

The following lemma shows that on the public π -calculus, bisimilarity is a quite discriminating relation.

Lemma 3. Given two public processes P and Q, if $P \sim Q$, then $\operatorname{fn}(P) = \operatorname{fn}(Q)$, size(P) = size(Q) and moreover P and Q have the same number of input (resp. output) prefixes. In particular, for P public, $P \sim \mathbf{0}$ implies $P \equiv \mathbf{0}$.

2.3 The Logic

Formulas of \mathcal{L} , the contextual spatial logic, are ranged over using $\mathcal{A}, \mathcal{B}, \ldots$, and are given by the following grammar:

 $\mathcal{A} \quad ::= \quad \top \quad \mid \quad \neg \mathcal{A} \quad \mid \quad \mathcal{A}_1 \land \mathcal{A}_2 \quad \mid \quad \diamond \mathcal{A} \quad \mid \quad \mathcal{A}_1 \triangleright \mathcal{A}_2 \quad \mid \quad \mathcal{A} \otimes n \quad \mid \quad \mathbf{M}n. \mathcal{A}.$

Name *n* is bound in $Un.\mathcal{A}$, and we let $fn(\mathcal{A})$ stand for the set of free names of \mathcal{A} . $\mathcal{A}_{\{n \leftarrow m\}}$ (resp. $\mathcal{A}_{\{n \leftarrow m\}}$) stands for the formula obtained by replacing (resp. permuting) all occurrences of *n* with *m* (resp. and *m*) in \mathcal{A} .

Definition 2 (Satisfaction in \mathcal{L}, Logical Equivalence). The judgement $P \models \mathcal{A}$, saying that process P satisfies formula \mathcal{A} , is defined as follows:

$$\begin{array}{l} -P \models \top always; \\ -P \models \neg \mathcal{A} \text{ iff not } P \models \mathcal{A} \text{ (also written } P \not\models \mathcal{A}); \\ -P \models \mathcal{A}_1 \land \mathcal{A}_2 \text{ iff } P \models \mathcal{A}_1 \text{ and } P \models \mathcal{A}_2; \\ -P \models \diamond \mathcal{A} \text{ iff there exists } P' \text{ s.t. } P \longrightarrow P' \text{ and } P' \models \mathcal{A}; \\ -P \models \mathcal{A}_1 \triangleright \mathcal{A}_2 \text{ iff for any } Q \text{ s.t. } Q \models \mathcal{A}_1, P | Q \models \mathcal{A}_2; \\ -P \models \mathcal{A} \otimes n \text{ iff } (\nu n) P \models \mathcal{A}. \\ -P \models \text{ Mn.} \mathcal{A} \text{ iff for any } m \text{ s.t. } m \notin \text{fn}(P) \text{ and } m \notin \text{fn}(\mathcal{A}), P \models \mathcal{A}_{\{n \leftrightarrow m\}}. \end{array}$$

TEAM LING

P and *Q* are logically equivalent, written $P =_{\mathcal{L}} Q$, iff for any formula \mathcal{A} , $P \models \mathcal{A}$ iff $Q \models \mathcal{A}$.

We will also make use in our constructions of the following derived formulas:

$$\begin{array}{cccc} \bot \stackrel{\mathrm{def}}{=} \neg \top & \mathcal{A}_1 \lor \mathcal{A}_2 \stackrel{\mathrm{def}}{=} \neg (\neg \mathcal{A}_1 \land \neg \mathcal{A}_2) & \Box \mathcal{A} \stackrel{\mathrm{def}}{=} \neg \Diamond \neg \mathcal{A} \\ \\ \mathcal{A}_1 \blacktriangleright \mathcal{A}_2 \stackrel{\mathrm{def}}{=} \neg (\mathcal{A}_1 \triangleright \neg \mathcal{A}_2) & \end{array}$$

The interpretation of \bot , \lor and \Box ('always') is standard. $P \models A_1 \blacktriangleright A_2$ iff there exists Q s.t. $Q \models A_1$ and $P|Q \models A_2$. Operators \triangleright and \blacktriangleright are right associative, and we define the following abbreviation: $\mathcal{A}_{\flat}^{\mathsf{h}}\mathcal{B} \stackrel{\text{def}}{=} \mathcal{A} \triangleright \mathcal{B}$, and $\mathcal{A}_{\flat}^{k+1}\mathcal{B} \stackrel{\text{def}}{=} \mathcal{A} \triangleright (\mathcal{A}_{\flat}^{k}\mathcal{B})$. We also set $\diamondsuit^{1}\mathcal{A} \stackrel{\text{def}}{=} \diamondsuit \mathcal{A}$ and $\diamondsuit^{k+1}\mathcal{A} \stackrel{\text{def}}{=} \diamondsuit(\diamondsuit^{k}\mathcal{A})$. A process P satisfies $\mathcal{A}_{\flat}^{k}\mathcal{B}$ iff P, put in parallel with k processes satisfying \mathcal{A} , satisfies \mathcal{B} . $P \models \diamondsuit^{k}\mathcal{A}$ iff P can perform k reductions and then satisfy \mathcal{A} .

Proposition 2. If $P \models A$, then for any $Q, P \equiv Q$ implies $Q \models A$.

This result implies that $\equiv \subseteq =_{\mathcal{L}}$, and that for any *P* and *A*, *P* $\models \Diamond \mathcal{A}$ iff $P \models \langle \tau \rangle \mathcal{A}$, where $\langle \tau \rangle$ is the HM modality corresponding to $\xrightarrow{\tau}$ [MPW93].

3 Expressiveness of the Logic

3.1 Auxiliary Formulas – Characterising Basic Processes

We start by some technical constructions to capture elementary π -calculus terms.

$$\begin{split} \operatorname{nil} &\stackrel{\operatorname{def}}{=} \Box \bot \land (\Box \bot \triangleright \Box \bot) \qquad [(1)] \stackrel{\operatorname{def}}{=} \Diamond \top \qquad [(k+1)] \stackrel{\operatorname{def}}{=} \Diamond \top \land \Box [(k)] \\ & \operatorname{atom} \stackrel{\operatorname{def}}{=} \Box \bot \land (\top \blacktriangleright \Diamond \operatorname{nil}) \qquad \operatorname{atom}(n) \stackrel{\operatorname{def}}{=} \operatorname{atom} \land (\operatorname{nil} \oslash n) \\ & *(n) \stackrel{\operatorname{def}}{=} \operatorname{Ma.} (\operatorname{atom}(n) \blacktriangleright \Diamond \operatorname{atom}(a)) \qquad \overline{n} \langle m \rangle \stackrel{\operatorname{def}}{=} *(n) \triangleright \Diamond \operatorname{atom}(m) \\ & n(_) \stackrel{\operatorname{def}}{=} \overline{n} \langle n \rangle \triangleright \Diamond \operatorname{nil} \qquad \operatorname{duo}(n,m) \stackrel{\operatorname{def}}{=} (\operatorname{atom}(m) \triangleright \Box \bot) \land (\overline{n} \langle n \rangle \triangleright \Diamond \overline{m} \langle m \rangle)) \\ & \operatorname{testOut}(n,m) \stackrel{\operatorname{def}}{=} \operatorname{Ma.} \operatorname{atom}(n) \blacktriangleright \Diamond \operatorname{duo}(a,m) \\ & \operatorname{testIn}(n,m,a) \stackrel{\operatorname{def}}{=} (\operatorname{atom}(a) \triangleright \Box \bot) \land (*(n) \triangleright \Diamond (\operatorname{atom}(m) \blacktriangleright \Diamond \overline{a} \langle a \rangle))) \end{split}$$

We briefly comment on these formulas. Using the strong interpretation of operator \diamond , we can capture the class of processes that are bisimilar to an atom:

formula atom says that these are processes that necessitate the addition of a context in order to evolve in one step of reduction to a process satisfying 0. We then distinguish between the input and output polarity by exploiting the ability for a process to interact on a received fresh channel, in formula *(n). This formula is then used to derive formulas to characterise the ~-class of some processes of sizes 1, 2 and 3. The following lemma states this formally:

Lemma 4. The above formulas have the following interpretation:

$$-P \models \mathsf{nil} iff P \sim \mathbf{0}.$$

- $-P \models [k]$ iff there is no reduction sequence $P = P_0 \longrightarrow P_1 \longrightarrow \dots \longrightarrow P_i$, with i < k, such that P_i cannot perform any \longrightarrow -transition.
- $-P \models \operatorname{atom}(n)$ iff $P \sim n(m)$ or $P \sim \overline{n}\langle m \rangle$, for some m; for formula atom, name *n* is not fixed.
- $-P \models *(n) \text{ iff } P \sim n(x).\overline{x}\langle y \rangle \text{ or } P \sim n(x).x(y) \text{ for some } y.$
- $-P \models \overline{n}\langle m \rangle$ iff $P \sim \overline{n}\langle m \rangle$, and $P \models n(_)$ iff $P \sim n(x)$ for some x.
- When $n \neq m$, $P \models duo(n,m)$ iff $P \sim n(x) \cdot \overline{m} \langle m \rangle$ for some $x \neq m$.
- $-P \models \mathsf{testOut}(n,m) \text{ iff } P \sim n(x).x(y).\overline{m}\langle m \rangle \text{ for some } y.$
- $-P \models \text{testln}(n, m, a) \text{ iff } P \sim \overline{n}(m) \cdot \overline{a}(a) \text{ when } a \neq n \text{ and } a \neq m.$

Proof (*Sketch*). The interpretation of formulas nil, [k], atom and atom(n) is easy. We sketch the proof of some of the other cases. We say that P is \sim -atomic if $P \models \mathsf{atom}$.

Suppose $P \models *(n)$, and take a s.t. $a \neq n$ and $a \notin fn(P)$. We put P in presence of a process Q bisimilar to $\overline{n}(m)$ or to n(m) for some m. Since P|Qreduces to a process that admits a as a free name and $a \notin fn(P)$, $a \in fn(Q)$, and thus $Q \sim \overline{n} \langle a \rangle$ because $a \neq n$. The reduction is necessarily an interaction between P and Q, since it leads to a term satisfying atom(a), and $n \neq a$. Hence

 $P \xrightarrow{n(a)} P'$ for some P', and $P'|\mathbf{0} \models \mathsf{atom}(a)$. As $a \notin \mathrm{fn}(P)$, we can conclude.

Suppose $P \models \overline{n}\langle m \rangle$. We put P in presence of a process Q satisfying *(n); the reduction step must be an interaction between P and Q, and P is \sim -atomic, otherwise we could not reach a ~-atomic process. Necessarily $P \sim \overline{n} \langle m \rangle$, because otherwise it could not react with Q and lead to a process satisfying atom(m).

Suppose $P \models duo(n, m)$, and $n \neq m$. P cannot interact at m and can receive n at n, leading to a process which is bisimilar to $\overline{m}(m)$. This is enough to conclude.

Suppose $P \models \mathsf{testOut}(n, m), a \notin \mathsf{fn}(P), a \neq n$, and $a \neq m$. We reason like in the case of *(n) to deduce that P is put in presence with a process bisimilar to $\overline{n}\langle a \rangle$. This implies that $P \xrightarrow{n(a)} P' \models \mathsf{duo}(a, m)$, and hence $P \sim \overline{n}\langle m \rangle . \overline{a}\langle a \rangle$.

3.2 **Detecting Barbs**

Although \mathcal{L} allows us to put a process in an arbitrary context built using parallel composition and restriction, what is missing to capture behavioural equivalence is the ability to perform instantaneous observations. We achieve this by introducing formulas to characterise barbs, i.e., the possibility for a term to offer an

interaction. We have not been able to define such formulas in the general case. Instead, our constructions depend on the size of the tested process, and are thus parametric over a natural number $k \ge 1$:

$$pol^{(k)}(a) \stackrel{\text{def}}{=} a(_)_{\triangleright}^{k} \diamondsuit^{k} \text{nil}$$

$$prefpol^{(k)}(n,a) \stackrel{\text{def}}{=} (atom(a) \triangleright \Box \bot) \land (\overline{n} \langle n \rangle \triangleright \Diamond pol^{(k)}(a))$$

$$\downarrow_{\overline{n}}^{(k)} \stackrel{\text{def}}{=} Ma.Mb. (duo(n,a) \triangleright prefpol^{(k)}(a,b) \triangleright a(_) \triangleright b(_)_{\triangleright}^{k} \diamondsuit \Diamond [\langle k \rangle])$$

$$\downarrow_{n}^{(k)} \stackrel{\text{def}}{=} Ma.Mb. (testln(n,n,a) \triangleright prefpol^{(k)}(a,b) \triangleright a(_) \triangleright b(_)_{\triangleright}^{k} \diamondsuit \Diamond [\langle k \rangle])$$

Lemma 5. Given a process P and an integer k, we have:

 $\begin{aligned} -P &\models \mathsf{pol}^{(k)}(a) \text{ iff } P \text{ is bisimilar to a term of the form } (\nu \tilde{c}) (\bar{a} \langle b_1 \rangle | \dots | \bar{a} \langle b_k \rangle), \\ \text{for some names } b_1, \dots, b_k, \tilde{c} \subseteq \{b_1, \dots, b_k\} \text{ and } a \notin \tilde{c}. \\ -P &\models \mathsf{prefpol}^{(k)}(n, a) \text{ iff } n \neq a \text{ and } P \sim n(x).P' \text{ with } P' \models \mathsf{pol}^k(a). \\ -P &\models \downarrow_{\overline{n}}^{(k)} \text{ iff } P \xrightarrow{\overline{n}(m)} P' \text{ for some } m, P', \text{ when } 2 * k > \mathsf{size}(P). \\ -P &\models \downarrow_n^{(k)} \text{ iff } P \xrightarrow{\overline{n}(m)} P', \text{ for some } m, P', \text{ when } 2 * k > \mathsf{size}(P). \end{aligned}$

Proof (sketch). We focus on formula $\downarrow_{\overline{n}}$. We first show that if $P \xrightarrow{\overline{n}\langle\langle m \rangle\rangle} P'$ for some m and P', then $P \models \downarrow_{\overline{n}}$. When P is put in parallel with the processes specified by the formula for $\downarrow_{\overline{n}}$, we can observe the following two reaction steps: P can interact with the process satisfying duo(n, a), thus liberating a process that can perform an output on a, which can in turn react with the term satisfying prefpol^(k)(a, b), yielding a state where formula $[\langle k \rangle]$ is satisfied (thanks to communications on b).

Suppose now $P \models \downarrow_{\overline{n}}$ and $\operatorname{size}(P) < 2 * k$. The scenario described by formula $\downarrow_{\overline{n}}$ expresses a property of the reductions of a process of the form $T = P|Q_1|Q_2|Q_3|Q_4$, where the Q_i s are tester processes specified by the formula:

- Q_1 can perform an input at n followed by an output at a.
- Q_2 starts by performing an input on a and then (independently from the received value) is liable to do k outputs at b.
- Q_3 just performs an input at a, while Q_4 can perform k inputs at b.

First observe that, since a and b are fresh, process $Q_1|Q_2|Q_3|Q_4$ cannot reduce on its own, and can only perform an input at n.

Then, consider a reduction $T \longrightarrow T'$, and suppose that it comes from a reduction $P \longrightarrow P'$, the Q_i s remaining inactive. We show that formula $\Diamond [\![k]\!]$ does not hold for T'. For this, we look for a term T'' s.t. $T' \longrightarrow T''$ and $T'' \models [\![k]\!]$.

- 1. Suppose the reduction to T'' results from P' performing a free output at n (the case where P' performs a bound output is treated similarly) and synchronising with Q_1 , then we obtain a process $T'' \equiv P''|Q_1'|Q_2|Q_3|Q_4$, where $P' \xrightarrow{\overline{n}(m)} P''$ for some m and $Q_1' \xrightarrow{\overline{a}(a)} \sim 0$. This entails that we can derive $T'' \longrightarrow U = P''|Q_2|Q_4$ from an interaction between Q_1' and Q_3 . As a result, process $Q_2|Q_4$ is stuck in all possible evolutions of U, since P'' does not know names a and b. So the only possible reductions of U are reductions resulting from P'' on its own. Since size(P) < 2 * k, we can conclude using Lemma 2 that $T'' \not\models [k]$.
- 2. Suppose the reduction to T'' results from P' performing a reduction step on its own, to a process P''. Then there are two cases:
 - (a) Either there exists R s.t. $P'' \longrightarrow^* R$ and $R \xrightarrow{\overline{n}(m)} R'$ for some m and R'. In this case, we reason as above to show that $R|Q_1|Q_2|Q_3|Q_4 \longrightarrow R'|Q_1'|Q_2|Q_3|Q_4 \longrightarrow R'|Q_2|Q_4$, and in the resulting state, process $Q_2|Q_4$ is stuck, which shows that $T'' \not\models [k]$. The case where R does a bound output at n is treated similarly.
 - (b) Either such an R does not exist, in which case $Q_1|Q_2|Q_3|Q_4$ is frozen in all possible evolutions of P'', and, since size(P) < 2 * k, $T' \neq [k]$.

So finally, there is no process T'' fulfilling the conditions stated above. This implies that no reduction of T involving only P can lead to a state where the formula is satisfied. So necessarily, P has to interact with $Q_1|Q_2|Q_3|Q_4$, which is possible only if P can perform an output at n.

The interpretation of formula \downarrow_n follows the same ideas, the testing process being specified using formula testln instead of duo.

When clear from the context, we will omit the superscript (k) in $\downarrow_n^{(k)}$, $\downarrow_{\overline{n}}^{(k)}$; we will do so in particular when P is fixed (cf. the proof of Lemma 8).

3.3 Characteristic Formulas for Public Processes

As remarked above, \mathcal{L} has the modality $\langle \tau \rangle$, due to the presence of constructor \diamond . We further have derivability of the following modalities, that will be useful below to define characteristic formulas.

Lemma 6 (Modality Formulas). The formulas above have the following interpretation, when size(P) < 2 * k:

$$- P \models \langle mn \rangle^{(k)} \mathcal{A} \text{ iff } P \xrightarrow{mn} P' \text{ and } P' \models \mathcal{A} \text{ for some } P'. \\ - P \models \langle \overline{m} \langle n \rangle \rangle^{(k)} \mathcal{A} \text{ iff } P \xrightarrow{\overline{m} \langle n \rangle} P' \text{ and } P' \models \mathcal{A} \text{ for some } P'.$$

TEAM LING

Proof (Sketch). In the case of $\langle mn \rangle^{(k)}$. \mathcal{A} , we put the candidate process in presence of a process bisimilar to $\overline{n}\langle m \rangle . \overline{a}\langle a \rangle | a(x)$, for *a* fresh. The remainder of the formula specifies that after two steps of reduction, the process must not exhibit a barb on *a* and must satisfy the continuation formula \mathcal{A} . This is possible only if a communication on *a* has happened, preceded by the output at *n*.

The formula for the free output modality follows similar ideas, the tester process being more complex due to the necessity to recognise two names in the prefix that is triggered. $\hfill \Box$

Remark 1. An important property of our constructions is that at the end of the 'experiment', no garbage process is left, so that we can go on with the satisfaction of A. This allows us to avoid using | in formulas like is done e.g. in [HLS03].

Remark 2 (Bound Output Modality). Although we have no formal proof for this, we believe that we cannot define a formula for the bound output modality in general in \mathcal{L} . Intuitively, the reason is that in order to define a formula $\langle \overline{n}(m) \rangle \mathcal{A}$, we should be able to impose satisfaction of \mathcal{A} under the restriction binding the extruded name, which is not possible (see also the extensions of \mathcal{L} in 5.2).

However, we can observe the ability for a process to perform a bound output:

Lemma 7. Given P, n and k such that $\operatorname{size}(P) < 2 * k$, there exists a formula $\langle \overline{n}(.) \rangle^{(k)}$ such that $P \models \langle \overline{n}(.) \rangle^{(k)}$ iff $P \xrightarrow{\overline{n}(m)} P'$ for some m and P'.

For lack of space, we do not present the proof of this result. The main idea is to express the fact that there is no way for a process coming from 'outside the tested process' to interact on the name received at n. We use for this some formulas whose interpretation contain a form of universal quantification on names (to give an idea, this is the case for example for formula atom introduced in 3.1).

The formulas given by Lemma 6 allow us to derive the following result.

Theorem 2 (Characterising Public Processes). For any public process P, there exists a formula F_P such that for any process Q, $Q \models F_P$ iff $P \sim Q$.

Proof (Sketch). We exploit the characterisation of ~ in Proposition 1, as well as Lemma 3 to simplify the formulas we manipulate. We define F_P by induction over the size of the transition system for \sim_s generated by *P*, using nil for the bisimilarity class of **0**. According to the result given in Proposition 1, we first pick *n* fresh names a_1, \ldots, a_n , where *n* is the number of input prefixes in *P*, and define $\mathcal{N} = \operatorname{fn}(P) \cup \{a_1, \ldots, a_n\}$. We define \mathcal{M} as the following set of actions:

$$\mathcal{M} \stackrel{\text{def}}{=} \{mn, m \in \mathcal{N}, n \in \mathcal{N}\} \cup \{\overline{m} \langle n \rangle, m \in \mathcal{N}, n \in \mathcal{N}\}.$$

We also set $Ac(P) = \{\mu, \exists P'. P \xrightarrow{\mu} P'\}$, and, for a free input or output action $\mu, P_{\mid \mu} = \{P'. P \xrightarrow{\mu} P'\}$. We then define:

$$\mathsf{F}_{P} \stackrel{\text{def}}{=} \mathsf{M}a_{1}\ldots \mathsf{M}a_{n} \cdot \bigwedge_{\mu \in \mathsf{Ac}(P)} \left[\mu\right]^{(k)} \cdot \left(\bigvee_{P' \in P_{|\mu}} \mathsf{F}_{P'}\right) \wedge \bigwedge_{\mu \in \mathcal{M} \setminus \mathsf{Ac}(P)} \neg \langle \mu \rangle^{(k)} \cdot \top,$$

where $[\mu]^{(k)}$. \mathcal{A} stands for $\neg \langle \mu \rangle^{(k)}$. $\neg \mathcal{A}$ and k satisfies $2 * k > \mathsf{size}(P)$.

TEAM LING

The construction of F_P is rather standard, and consists in describing the transitions a state can make by expressing the possible actions and their continuations, as well as those actions that cannot be performed.

Remark 3. The characteristic formulas we define are valid for the whole calculus, and in particular they are also satisfied by all processes with restriction that are bisimilar to a public process. The logic also allows us to characterise some processes outside this class, as illustrated by the following formula

$$\langle \overline{a}(.) \rangle^{(k)} \wedge \operatorname{Wr.} \left(\mathsf{F}_{a(x).x(y).\overline{r}\langle y \rangle} \triangleright \diamondsuit \overline{r} \langle b \rangle \right),$$

which captures the processes bisimilar to $(\nu c) (\overline{a} \langle c \rangle | \overline{c} \langle b \rangle)$ for $k \geq 2$ (this is the case e.g. for $(\nu c) \overline{a} \langle c \rangle . \overline{c} \langle b \rangle$). We have not been able to define characteristic formulas for the whole calculus, though, and do not believe that this would be feasible along the lines of the constructions presented above.

Remark 4 (On the Role of \mathbf{N}). We could get rid of \mathbf{N} in the constructions we have presented. This is possible by defining a formula $\mathsf{unactive}^{(k)}(n)$, that says that name n is not liable to be used in the first k interactions of a given process P provided n is not sent to P (we can easily express (dis)equality of names in \mathcal{L}). Note that this property is different from being a fresh name for P, as a process bisimilar to $\mathbf{0}$ can have free occurrences of names. Intuitively, to obtain extensionality (Theorem 4 below), we only need to be able to pick enough 'unactive names' to build the characteristic formula for a given public process. Without having checked formally that this is the case, we do believe that our main result can be proved in a logic without \mathbf{N} .

4 Extensionality

We now show that $=_{\mathcal{L}}$ coincides with ~.

Theorem 3 (Behavioural Implies Logical). $P \sim Q$ implies $P =_{\mathcal{L}} Q$.

Proof (Sketch). We prove by structural induction on \mathcal{A} that whenever $P \sim Q$ and $P \models \mathcal{A}$, we have $Q \models \mathcal{A}$. The cases corresponding to the adjunct operators \triangleright and \otimes follow from congruence properties of \sim w.r.t. parallel composition and restriction, respectively (see [SW01]).

Lemma 8 (Characteristic Formulas for Barbed Bisimilarity). Given a process P and a finite set of names \mathcal{N} such that $\operatorname{fn}(P) \subseteq \mathcal{N}$, there exists a formula $\mathcal{B}_{\mathcal{N},P}$ such that for any Q such that $\operatorname{fn}(Q) \subseteq \mathcal{N}$, $Q \models \mathcal{B}_{\mathcal{N},P}$ iff $P \simeq Q$.

Proof. The formula is defined by induction on the size of *P* as follows:

$$\mathsf{B}_{\mathcal{N},P} \stackrel{\mathrm{def}}{=} \bigwedge_{n \in \mathcal{N}. P \downarrow_{n}} \downarrow_{n} \land \bigwedge_{n \in \mathcal{N}. P \downarrow_{\overline{n}}} \downarrow_{\overline{n}} \land \bigwedge_{n \in \mathcal{N}. P \downarrow_{n}} \neg \downarrow_{n} \land \bigwedge_{n \in \mathcal{N}. P \downarrow_{\overline{n}}} \neg \downarrow_{\overline{n}} \\ \land \bigwedge_{P' \in \{P'. P \longrightarrow P'\}_{/\overline{n}}} \diamondsuit \mathsf{B}_{\mathcal{N},P'} .$$

In this formula, $S_{/=}$ stands for the quotient of S modulo \equiv . The above conjunction is finite because N is finite and reduction is image-finite up to \equiv ([SW01]). The definition is well-formed because P' is smaller (in the senze of size) than P in the recursive calls.

By definition of \simeq , $Q \models \mathsf{B}_{\mathcal{N},P}$ iff $P \simeq Q$, as long as all free names of Q are inspected by formula $\mathsf{B}_{\mathcal{N},P}$, which is guaranteed by the condition $\mathrm{fn}(Q) \subseteq \mathcal{N}$.

Theorem 4 (Logical Implies Behavioural). $P =_{\mathcal{L}} Q$ implies $P \sim Q$.

Proof. Suppose $P \not\sim Q$; by Theorem 1, $P \not\simeq Q$, i.e., $P | R \not\simeq Q | R$ for some R.

Write using Lemma 2 $R \sim (\nu \tilde{n}) R_0$, where R_0 is public and $\tilde{n} \cap \text{fn}(P) = \tilde{n} \cap \text{fn}(Q) = \emptyset$. We have that $\sim \simeq \simeq \simeq \simeq \simeq$, and hence $(\nu \tilde{n}) (P|R_0) \not\cong (\nu \tilde{n}) (Q|R_0)$.

Take $\mathcal{N} = \operatorname{fn}(P) \cup \operatorname{fn}(Q) \cup \operatorname{fn}(R)$, and define $\mathcal{A} \stackrel{\text{def}}{=} \operatorname{B}_{\mathcal{N},(\boldsymbol{\nu}\widetilde{n})(P|R_0)} \otimes \widetilde{n}$. Observe that we have $P \models \operatorname{F}_{R_0} \triangleright \mathcal{A}$. Suppose now $Q \models \operatorname{F}_{R_0} \triangleright \mathcal{A}$, this means that for all R_1 such that $R_0 \sim R_1$, $(\boldsymbol{\nu}\widetilde{n})(Q|R_1) \models \mathcal{A}$, which entails by Lemma 8 that $(\boldsymbol{\nu}\widetilde{n})(P|R_0) \simeq (\boldsymbol{\nu}\widetilde{n})(Q|R_1)$. $R_0 \sim R_1$ implies $(\boldsymbol{\nu}\widetilde{n})(Q|R_1) \sim (\boldsymbol{\nu}\widetilde{n})(Q|R_0)$, and hence, since $\sim \subseteq \simeq$, and by transitivity of \simeq , we obtain $(\boldsymbol{\nu}\widetilde{n})(P|R_0) \simeq (\boldsymbol{\nu}\widetilde{n})(Q|R_0)$, a contradiction. So $Q \models \operatorname{F}_{R_0} \triangleright \mathcal{A}$, and finally $P \neq_{\mathcal{L}} Q$.

Note that the proof above exploits the two presentations of \sim : characteristic formulas for public processes are derived using the labelled transition system, while the overall structure of the proof follows the definition of \simeq .

5 Variants and Extensions of \mathcal{L}

5.1 Changing the Primitive Observation

The most tedious constructions in Section 3 are the formulas to detect barbs (cf. Theorem 5). We consider here a variant of \mathcal{L} , called \mathcal{L}^{\ddagger} , in which we remove \otimes and add a primitive formula n^{\ddagger} , whose satisfaction is defined by $P \models n^{\ddagger}$ iff $(P \downarrow_n \text{ or } P \downarrow_{\overline{n}})$. With respect to \mathcal{L} , \mathcal{L}^{\ddagger} allows one to build less contexts, while providing the ability to perform instantaneous observations independently from the size of the tested process.

We first remark that the only place where \otimes is used in the formulas presented in Section 3 is in the definition of atom(n), that can be rewritten in \mathcal{L}^{\uparrow} as follows:

$$\mathsf{atom}(n) \stackrel{\mathrm{def}}{=} \mathsf{atom} \wedge n^{1}$$
 .

To show that logic \mathcal{L}^{\ddagger} induces an equivalence that also coincides with ~, the completeness proof of Section 4 has to be adapted. The main point is to observe that testing for $\dot{\simeq}$ against *public* processes is enough to get the same discriminative power as \simeq , as expressed by the following lemma.

Lemma 9. Define $P \simeq_p Q$ iff for any R public, $P | R \simeq Q | R$. Then $\simeq_p = \simeq$.

This result allows us to replay the proof of Theorem 4 in the case of \mathcal{L}^{\ddagger} without using \heartsuit , and we have:

Theorem 5. Two processes are logically equivalent for \mathcal{L}^{\ddagger} iff they are bisimilar.

5.2 Enriching \mathcal{L} with Intensional Operators

Our main result, given by Theorems 3 and 4, isolates the extensional subset of spatial logics for concurrency. To explore the spectrum between extensional and intensional spatial logics, we now consider enrichments of \mathcal{L} with (some of the) intensional operators 0, | and \mathbb{B} .

Observing Emptyness. Formula 0 gives us an elementary form of observation: in $\mathcal{L} \cup \{0\}$ (using an obvious notation), we can detect garbage, in the sense that we can use 0 to separate for example processes **0** and $(\nu n)\overline{n}\langle m\rangle$. *P* (which are bisimilar). As a consequence, using 0 instead of nil in the constructions of Section 3, we can define characteristic formulas for 'minimal-size public processes', i.e., characteristic formulas up to ~ that are satisfied only by public processes.

Separating. In $\mathcal{L} \cup \{|\}$, we get a finer equivalence than ~. In particular, we have: Lemma 10. In $\mathcal{L} \cup \{|\}$, logical equivalence on public terms coincides with \equiv .

On the full calculus, it would be interesting to study the relationship with *distributed bisimulation* [CH89], a behavioural equivalence that is able to separate for example $\overline{m}\langle n \rangle . \overline{m}\langle n \rangle$ and $\overline{m}\langle n \rangle | \overline{m}\langle n \rangle$. In presence of restriction, though, the definition of distributed bisimulation is rather complex, even for CCS [Kie89].

Revealing Names. The results in [HLS03] show that revelation brings a lot of expressiveness to the logic. In $\mathcal{L} \cup \{\mathbb{P}\}$, formula 0 is derivable, as well as a formula to test the free occurrence of a name at any depth in a process:

 $0 \stackrel{\text{def}}{=} \operatorname{nil} \wedge \neg \operatorname{Ma.a}_{\mathbb{R}} \neg \operatorname{nil} \qquad \operatorname{free}(n) \stackrel{\text{def}}{=} \neg n \mathbb{R} \top$

(the second formula is from [CG01b] — note that in the static case, free(n) can be defined using only \otimes [CG04]). Having the ability to observe under restrictions, we can define a modality formula for bound output:

Lemma 11. In $\mathcal{L} \cup \{ \mathfrak{B} \}$, given a formula \mathcal{A} and a process P such that size(P) < 2 * k - 1, there exists a formula $\langle \overline{m}(n) \rangle^{(k)} \mathcal{A}$ such that $P \models \langle \overline{m}(n) \rangle^{(k)} \mathcal{A}$ iff $P \xrightarrow{\overline{m}(n)} P'$ and $P' \models \mathcal{A}$ for some P'.

We believe that characteristic formulas up to ~ for the whole calculus are definable in this enriched logic. (a ctually gives us a greater precision. Indeed, the induced logical equivalence is rather fine-grained, but we have not been able to provide a precise characterisation of it. For example, $(\nu c) \overline{a} \langle c \rangle . \overline{c} \langle b \rangle$ can be separated from $(\nu c) (\overline{a} \langle c \rangle | \overline{c} \langle b \rangle)$, while $\overline{m} \langle n \rangle . \overline{m} \langle n \rangle$ and $\overline{m} \langle n \rangle | \overline{m} \langle n \rangle$ are equivalent in $\mathcal{L} \cup \{\mathbb{R}\}$.

Combining Intensional Observations. In $\mathcal{L} \cup \{0, |\}$, on the other hand, the logic separates the two latter processes while equating the first two. Finally, logic $\mathcal{L} \cup \{|, \mathbb{R}\}$ is intensional: logical equivalence coincides with \equiv . This can be shown by adapting the proofs in [HLS02] using the constructions of the present paper.

The following graph sums up the observations made above. Vertices contain relations, and two relations situated on the same vertex coincide ($=_{\mathbb{L}}$ stands for the equality induced by the logic L). An arrow between two edges represents strict inclusion between the corresponding relations, and unrelated vertices correspond to uncomparable relations. More detailed explanations are given in[Hir04].



6 Conclusion

We have defined a spatial logic \mathcal{L} , and shown that the induced logical equivalence coincides with bisimilarity. We can remark that while HM logic and \mathcal{L} induce the same equivalence on processes, model-checking in \mathcal{L} seems a priori much more difficult, due to the presence of \triangleright . We can also remark that \mathcal{L} has a restricted set of operators, and is minimal in the sense of [Loz03]: getting rid of a connective of \mathcal{L} hinders the expressive and separative powers of the logic.

The observations provided in \mathcal{L}^{\ddagger} suggest that this logic is to barbed equivalence what Hennessy-Milner logic is to bisimilarity. The constructions in 3.2 show that, to some extent, \mathcal{L} has the ability to express the observations of \mathcal{L}^{\ddagger} . We do not see how the converse could hold, i.e. how hiding could be expressed within \mathcal{L}^{\ddagger} . At least we believe there is no compositional encoding that could translate a formula of the form $\mathcal{A} \otimes n$ into \mathcal{L}^{\ddagger} . A perhaps more interesting question would be to find out whether \otimes is eliminable in the logic resulting from the 'union' of \mathcal{L} and \mathcal{L}^{\ddagger} , along the lines of adjunct elimination in [Loz03, Loz04, DGG04].

In contrast with existing spatial logics, that all include intensional operators, satisfaction in \mathcal{L} is defined with no direct reference to structural congruence. It would be interesting to look for a way to recover some of the separating power of intensional spatial logics while keeping logical equivalence close to some existing, previously defined, relation on processes. In another direction, we would like to see whether we can combine the ideas presented here with the constructions defined in [CL04], which would mean studying $\mathcal{L} \setminus \{ \odot \}$, a contextual logic with no reference to names. Logical equivalence is up to name permutation in the name-free logic of [CL04]: we conjecture that $=_{\mathcal{L} \setminus \{ \odot \}}$ is bisimilarity up to name permutation. This would mean that extensionality of spatial logics is brought by \triangleright , an observation that is already suggested by the results of this paper.

Finally, our techniques do not apply directly if we renounce to the strong interpretation we adopt for \diamond (which we use here to count, in some way): studying logical equivalence in a 'weak version' of \mathcal{L} represents a challenging question.

Acknowledgements. We would like to thank Étienne Lozes, Davide Sangiorgi and Luís Caires for inspiring discussions about the results presented in this paper. This work has been supported by european FET - Global Computing project PROFUNDIS and by the french ACI GEOCAL.

References

[CC01]	L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part I). In <i>Proc. of TACS'01</i> . LNCS. Springer Verlag, 2001.
[CG00]	L. Cardelli and A. Gordon. Anytime, Anywhere, Modal Logics for Mobile Ambients. In <i>Proc. of POPL'00</i> , pages 365–377. ACM Press, 2000.
[CG01a]	L. Cardelli and G. Ghelli. A Query Language Based on the Ambient Logic. In <i>Proc. of ESOP'01</i> , volume 2028 of <i>LNCS</i> , pages 1–22. Springer Verlag, 2001 invited paper
[CG01b]	L. Cardelli and A. Gordon. Logical Properties of Name Restriction. In <i>Proc. of TLCA'01</i> , volume 2044 of <i>LNCS</i> . Springer Verlag, 2001
[CG04]	G. Conforti and G. Ghelli. Decidability of Freshness, Undecidability of Revelation. In <i>Proc. of FOSSACS'04</i> , volume 2987 of <i>LNCS</i> , pages 105–120. S. J. V. J. 2004
[CH89]	I. Castellani and M. Hennessy. Distributed Bisimulations. J. ACM, 36(4):887-911 1989
[CL04]	L. Caires and E. Lozes. Elimination of Quantifiers and Undecidability in Spatial Logics for Concurrency, this volume, 2004
[DGG04]	A. Dawar, P. Gardner, and G. Ghelli. Games for the Ambient Logic. draft, 2004
[DLM04]	S. Dal Zilio, D. Lugiez, and C. Meyssonnier. A Logic You Can Count on. In <i>Proc. of POPL 2004</i> , ACM Press, 2004.
[Hir04]	D. Hirschkoff. An Extensional Spatial Logic for Mobile Processes. Technical report. LIP - ENS Lyon, 2004. to appear
[HLS02]	D. Hirschkoff, E. Lozes, and D. Sangiorgi. Separability, Expressiveness and Decidability in the Ambient Logic. In <i>Proc. of LICS'02</i> , pages 423–432. IEEE Computer Society, 2002.
[HLS03]	D. Hirschkoff, E. Lozes, and D. Sangiorgi. Minimality Results for the Spatial Logics. In <i>Proc. of FSTTCS'03</i> , volume 2914 of <i>LNCS</i> , pages 252–264. Springer Verlag, 2003.
[Kie89]	A. Kiehn. Distributed Bisimulations for Finite CCS. Technical Report 7/89. University of Sussex, 1989.
[Loz03]	E. Lozes. Adjunct Elimination in the Static Ambient Logic. In Proc. of EXPRESS'03, ENTCS. Elsevier, 2003.
[Loz04]	E. Lozes. Separation logic preserves the expressiveness of classical logic. In <i>Proc. of SPACE'04</i> , 2004.
[MPW93]	Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. <i>Theoretical Computer Science</i> , 114(1):149–171, 1993.
[Rey02]	J. Reynolds. Separation logic: a logic for shared mutable data structures. In <i>Proc. of LICS'02</i> . IEEE Computer Society, 2002.
[San01]	D. Sangiorgi. Extensionality and Intensionality of the Ambient Logic. In <i>Proc. of 28th POPL</i> , pages 4–17. ACM Press, 2001.
[SW01]	D. Sangiorgi and D. Walker. The π -calculus: a Theory of Mobile Processes. Cambridge University Press, 2001.

Timed vs. Time-Triggered Automata

Pavel Krčál¹, Leonid Mokrushin¹, P.S. Thiagarajan², and Wang Yi¹

¹ Dept. of Information Technology, Uppsala University, Sweden {pavelk, leom, yi}@it.uu.se

² Dept. of Computer Science, National University of Singapore thiagu@comp.nus.edu.sg

Abstract. To establish a semantic foundation for the synthesis of executable programs from timed models, we study in what sense the timed language (i.e. sequences of events with real-valued time-stamps) of a timed automaton is recognized by a digital machine. Based on the noninstant observability of events, we propose an alternative semantics for timed automata. We show that the new semantics gives rise to a natural notion of digitalization for timed languages. As a model for digital machines we use time-triggered automata – a subclass of timed automata with simplified syntax accepting digitalized timed languages. A timetriggered automaton is essentially a time table for a digital machine (or a digital controller), describing what the machine should do at a given time point, and it can be easily transformed to an executable program. Finally, we present a method to check whether a time-triggered automaton recognizes the language of a timed automaton according to the new semantics.

1 Introduction

Timed automata [AD94] have been recognized as a basic model for real time systems. A number of tools based on timed automata have been developed (e.g. [BDM⁺98,LPY97]), and applied successfully to model and verify industrial systems (e.g. [BGK⁺96,DY00]). A recent line of work uses timed automata for the schedulability analysis of tasks in real time systems [FPY02,KY04,WH04]. The main idea here is to use timed automata to describe the arrival patterns of external events triggering real-time tasks. One implicit assumption in this line of work and its extensions to synthesize executable code [AFP⁺02] is that the arrival sequences of events described by the automaton can be admitted *instantly* by the interrupt-processing system of the given platform. However, this is not realistic in realizations involving a digital computer driven by a system clock with a fixed granularity of time.

Therefore, we wish to study the notion of timely executions of task arrival patterns in settings where the implementation model is explicitly clock-driven. Time-triggered architecture [KB01,Kop98] is one such well-recognized implementation paradigm for the design of safety-critical real-time systems, a programming language Giotto [HKH03] provides an abstract model for implementation

© Springer-Verlag Berlin Heidelberg 2004

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 340-354, 2004.

of time-triggered systems. In a time-triggered system, computation steps are controlled by one global clock; at each time moment only actions enabled by the clock value can be taken. Thus, computations are time-deterministic in such systems.

We present a (finite state) model of computations running on time-triggered architectures called *time-triggered automata* accepting digitalized versions of timed languages. A time-triggered automaton may take transitions (to consume external events) only at integer time points determined by periodic and nonperiodic timing constraints denoted a(n) and a[n] respectively. A transition labeled with a(n) means that every n time units, it should check if a has occurred; a transition labeled with a[n] means that in n time units from now, it should check if a has occurred. Our time-triggered automata capture only the basic aspect of time-triggered architectures. In the present setting they could as well be called "digital automata". However, we consider time-triggered architectures and protocols to be important and the model we propose here is a simple but potentially useful first step towards formalizing this framework in an automata theoretic setting. A time-triggered automaton is essentially a timetable for a digital controller, describing what the controller should do at a given time point, and it can be easily transformed to an executable program. Time-triggered automata can be viewed as special kinds of timed automata. However, their intended role is to capture the behavior of digital machines and their syntax is geared towards achieving this in a transparent way.

Our main goal is to semantically tie together the timed-automata based specifications to the time-triggered-automaton based implementations. We do so by associating a deadline parameter ϵ with the timed automaton. The idea is that there is a buffer¹ into which the event arrivals generated by the timed automaton will be put and each such event will be observable for ϵ time units from the time it was released (i.e. put into the buffer). A correct implementation mechanism is then required to pick up each event before its deadline expires. The parameter ϵ associated with the timed automaton and the unit of time associated with the time-triggered automaton are with reference to the same global time scale. Further, both types of automata share a common alphabet of events. These two components provide the handle for semantically relating the language of the timed automaton to that of the time-triggered automaton.

More specifically, given a timed automaton TA and the deadline parameter ϵ we extract the ϵ -digitalized timed language of TA. In loose terms, this language is obtained by taking a timed trace of TA and converting the timestamp t of each event into an integer-valued time k_t provided $k_t - t$ does not exceed ϵ . This captures the idea that an event released at time t can be, in principle, picked up at integer time k_t before the deadline associated with this release has expired.

We then associate a digitalized timed language with the corresponding timetriggered automaton TTA and formulate the notion of TTA correctly accepting

¹ For ease of presentation we consider the buffer to be FIFO, but the same ideas apply also to random access buffer modeled as a multiset.

 ϵ -digitalization of the timed language of TA. This notion is intended to capture the processing by TTA of the request patterns generated by TA. Each event generated by TA will be put in the buffer with an associated timer set to ϵ . At integer time points dictated by its transition relation, the TTA will pick up a specified sequence of events from the buffer provided none of these events have an expired timer. This whole process fails anytime an event misses its deadline before it gets picked by the TTA. In order to simplify the presentation we have identified the servicing of the request represented by an event with being picked up by the TTA in time. In other words, we have abstracted away the tasks associated with the events and their timely executions on the chosen platform which may be limited resource-wise. Further, we assume the same fixed observability period (i.e. deadline) for each event.

Finally, we show that it is possible to effectively decide whether a TTA correctly services the request patterns generated by the TA. The proof technique is an adaptation of the timed automaton based technique used in [FPY02] for solving schedulability problems.

1.1 Related Work

In the literature on timed automata, there has been a considerable amount of work related to discretization issues, e.g. [HMP92,LY97,GHJ97] where just the timed traces with integral timestamps of events are used to characterize the real time semantics, and to identify (digitizable) dense-time properties that can be checked based on the integral traces. A recent study on digitalization is [OW03] which is also a very good source for other related work. Despite the overlap in terminology the key difference is that here we do not consider the *actual* timed traces of a timed automaton. Rather, we study in what sense the timed traces of an automata can be recognized by a digital computer (i.e. a digital controller). The controller should consume (or record) all the timed traces of a timed automaton, not only those in which the timestamps of events are integers. Due to the non-instant observability of events, this consumption can take place at any integer time point shortly (within ϵ time units) after the event has occurred.

The idea of checking whether all events generated by an environment are picked-up by a synchronous program sufficiently fast has been also studied in [BPS00,BCP+01,CPP+01,STY03]. Here authors use ESTEREL programs annotated with temporal constraints as event handlers to consume events. A timed automata model of the application can be automatically generated out of such a program. Then the behavior of the model is checked against the environment (modeled again in ESTEREL with added non-deterministic primitive and then translated to a timed automaton). The presented methodology has been implemented in the toolTAXYS [CPP+01] and successfully used for industrial applications. Our focus here is on the basic semantic issues that arise in the interaction between timed automata and digital machines and our study is carried out in a language-independent automata theoretic setting. We consider timed automata as timed specifications and study in what sense such a specification involving dense time is implemented by a digital machine that is described as

time-triggered automaton. For $\epsilon = 1$ our correctness checking coincides with checking of throughput constraint satisfaction described in [CPP+01].

Recently, a new semantics for timed automata geared towards their implementation was proposed in [WDR04]. This work is also based on non-instant observability of events. Its main goal is to detect timed automata where two consecutive actions are taken shortly one after another. When this is not the case, the timed automaton can be implemented as an event-driven controller using sampling. In contrast with our semantics, this work deals with real timed traces. Also the implementation simulates timed automaton specification of a controller whereas in our case there is no timed automata model of a controller (in our case, the environment in the only specification).

The rest of the paper is organized as follows: Section 2 contains a brief introduction to timed automata and a detailed description of the ϵ -semantics and gives two alternative formalizations of digitalization. The syntax and semantics of time-triggered automata is defined in Section 3. Section 4 presents a correctness criterion for time-triggered automata and our technique for verifying correctness. Section 5 concludes the paper.

2 Timed Automata: ϵ -Semantics and Digitalization

In the following, we briefly introduce the standard semantics for timed automata. To capture the non-instant observability of timed events, we present a new semantics for timed automata. It gives rise to a simple and natural notion of digitalization for timed languages.

2.1 Timed Automata

A timed automaton [AD94] is a standard finite-state automaton extended with a finite collection of real-valued clocks. Events are accepted by an automaton at real-time points satisfying given timing constraints. Assume a set of real-valued clocks denoted by \mathcal{C} and a finite alphabet Σ whose elements represent events.

Definition 1. A timed automaton \mathcal{A}_{TA} is a tuple $\langle \mathcal{C}, \Sigma, N, l_0, E \rangle$ where

- -C is a set of real-valued clocks,
- $-\Sigma$ is a finite alphabet of events,
- -N is a finite set of locations,
- $-l_0 \in N$ is the initial location, and
- $-E \subseteq N \times \Phi(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times N$ is the set of edges (describing possible transitions).

The set $\Phi(\mathcal{C})$ of clock constraints ϕ is defined as a set of conjunctive formulas of atomic constraints in the form: $x_i \sim m$ or $x_i - x_j \sim n$ where $x_i, x_j \in \mathcal{C}$ are clocks, $\sim \in \{\leq, <, \geq, >\}$, and m, n are natural numbers. A clock valuation $u \in [\mathcal{C} \to \mathbb{R}_{\geq 0}]$ is a function mapping clocks to non-negative real numbers. We use u + t to denote the clock assignment which maps each clock x to the value u(x) + t, and $u[r \mapsto 0]$ for $r \subseteq C$ to denote the clock assignment which maps each clock in r to 0 and agrees with u for the other clocks (i.e. $C \setminus r$). An edge (l_1, ϕ, e, r, l_2) represents a transition from location $l_1 \in N$ to location $l_2 \in N$ accepting an input symbol (we will call it an *event*) $e \in \Sigma$, and resetting clocks in $r \subseteq C$ to zero, if the current values of clocks satisfy ϕ .



Fig. 1. An example timed automaton

An example timed automaton is shown in Figure 1. It consists of the set of locations $N = \{l_0, l_1\}$ where l_0 is the initial location, the set of clocks $\mathcal{C} = \{x\}$, the alphabet of events $\Sigma = \{a, b, c\}$, and the set of clock constraints $\Phi(\mathcal{C}) = \{x \ge 4, x \le 4, x > 3, x < 8\}$.

A timed event is a pair (t, e), where $e \in \Sigma$ is an event accepted by \mathcal{A}_{TA} after $t \in \mathbb{R}_{\geq 0}$ time units since \mathcal{A}_{TA} has been started. This absolute time t is called a *timestamp* of the event e. A *timed trace* is a (possibly infinite) sequence of timed events $\xi = (t_1, e_1)(t_2, e_2)$... over events $e_1, e_2, ...$ associated with the corresponding timestamps $t_1, t_2, ...$ where $\forall i \geq 1$: $t_i \leq t_{i+1}$.

Definition 2. A run of a timed automaton $\mathcal{A}_{TA} = \langle \mathcal{C}, \Sigma, N, l_0, E \rangle$ over a timed trace $\xi = (t_1, e_1)(t_2, e_2)(t_3, e_3)...$ is a (possibly infinite) sequence of the form

$$(l_0, u_0) \xrightarrow{e_1} (l_1, u_1) \xrightarrow{e_2} (l_2, u_2) \xrightarrow{e_3} \dots$$

where $l_i \in N$, u_i is a clock valuation, satisfying the following conditions:

- $-u_0(x)=0$ for all $x \in \mathcal{C}$.
- for all $i \ge 1$, there is an edge $(l_{i-1}, \phi_i, e_i, r_i, l_i)$ such that $(u_{i-1} + t_i t_{i-1})$ satisfies ϕ_i and $u_i = (u_{i-1} + t_i - t_{i-1})[r_i \mapsto 0]$.

The timed language $L(\mathcal{A}_{TA})$ over alphabet Σ is the set of all timed traces ξ for which there exists a run of \mathcal{A}_{TA} over ξ .

For example, the timed automaton shown in Figure 1 can run over the following timed trace: $\xi = (0.5, a)(4.5, c)(7.6, b)(7.9, a) \dots$

2.2 The ϵ -Semantics and Digitalization

One can interpret a timed automaton as the model of an environment producing events. For example, at some real-time points a plant can produce events to which a controller should respond. It is a natural assumption that each such event remains observable for a short time interval during which it can be picked up by a controller. However, the non-instant observability of events is not reflected in the standard semantics for timed-automata. We introduce the ϵ -semantics to capture the fact that each event remains observable for ϵ time units after its occurrence.

Assume that an event occurring at time t is put in a communication buffer between the environment and the controller and remains there until consumed by the controller or expired at $t + \epsilon$. We introduce a queue q to represent the buffer between the environment and the controller. Elements of q are pairs (e, δ) where $e \in \Sigma$ is an event and $\delta \in \mathbb{R}$ is the relative deadline of e in q. We denote q :: e the queue q with a pair (e, ϵ) inserted in the back of it, and $q \setminus e$ the queue q with (e, δ) removed from it where (e, δ) is the head of the queue q. We write q - d for the queue q in which deadlines in all pairs are decreased by d. The states of a timed automaton in the ϵ -semantics are in the form: (l, u, q, τ) where

- -l is the current location,
- -u is the current clock valuation,
- -q is the current event queue, and
- $-\tau$ is the current time i.e. the global time elapsed since the automaton has been started in the initial state.

For simplicity, we assume that all events are associated with a fixed constant $\epsilon \in \mathbb{N}_{>0}$ (by $\mathbb{N}_{>0}$ we denote the set of positive natural numbers). However, the setting can be easily extended to allow the constant to be a positive rational number.

Definition 3. Let $\epsilon \in \mathbb{N}_{>0}$ be a positive natural number. The ϵ -semantics of a timed automaton $\langle \mathcal{C}, \Sigma, N, l_0, E \rangle$ with initial state (l_0, u_0, q_0, τ_0) , where q_0 is an empty queue and $\tau_0 = 0$, is a labeled transition system S^{ϵ} defined by the following rules:

- -Consumption: $(l, u, q, \tau) \xrightarrow{e} (l, u, q \setminus e, \tau)$ if $(e, \delta) = \text{Head}(q)$ such that $\epsilon \ge \delta > 0$,
- Production: $(l_1, u_1, q, \tau) \longrightarrow (l_2, u_2, q :: e, \tau)$ if $\exists (l_1, \phi, e, r, l_2) \in E$ such that u_1 satisfies ϕ and $u_2 = u_1[r \mapsto 0]$.
- Delay: $(l, u, q, \tau) \xrightarrow{d} (l, u + d, q d, \tau + d)$.

If an event has not been consumed after ϵ time units by means of consumption transition then no other consequent event would be consumed later. Such a state is considered as a failure of a controller to consume all events. Note also that environments are allowed to produce infinitely many events in a finite time. Naturally, a controller for such environments cannot be implemented.

In order to describe how events produced by a timed automaton are consumed by a digital machine, we introduce a notion of a digitalized timed trace. In the ϵ semantics several events can be consumed at the same time but only in the order they have been produced, i.e. the controller cares about dependencies between events.

Definition 4. An ϵ -digitalized timed trace ξ^{ϵ} in S^{ϵ} is a (possibly infinite) sequence $(t_1^{\epsilon}, e_1), (t_2^{\epsilon}, e_2), (t_3^{\epsilon}, e_3), \ldots$ such that there exists a path in S^{ϵ} where
e_1, e_2, e_3, \ldots are labels on the consumption transitions in the order in which they occur on the path, $t_1^{\epsilon}, t_2^{\epsilon}, t_3^{\epsilon}, \ldots$ are their absolute timestamps respectively, and $t_i^{\epsilon} \in \mathbb{N}_{>0}$ for all *i*.

The ϵ -digitalized timed language $L^{\epsilon}(\mathcal{A}_{TA})$ over alphabet Σ according to the ϵ -semantics is the set of all ϵ -digitalized timed traces ξ^{ϵ} in \mathcal{S}^{ϵ} for \mathcal{A}_{TA} .

We shall see (in the following subsection) that for $\epsilon > 1$ each run of \mathcal{A}_{TA} can have several corresponding ϵ -digitalized timed traces in which the distance between the real-valued timestamp and corresponding digitalized timestamp for each event is limited by ϵ . For the case when $\epsilon = 1$, there is only one such ϵ -digitalized timed trace for each run of \mathcal{A}_{TA} . This is a useful property of our notion of digitalization. It means that any sequence of events with timestamps in the standard semantics will be caught by at least one digitalized trace. This enables to formulate the correctness criterion as a language inclusion property.

2.3 An Alternative Characterization

We present an alternative characterization of the ϵ -digitalized timed language for timed automata. This characterization establishes a connection between a timed trace and its ϵ -digitalized versions. In the following we use rounded-up time points [t] where [t] denotes the least integer such that $t \leq [t]$.

Definition 5. For a timed trace $\xi = (t_1, e_1)(t_2, e_2)(t_3, e_3) \dots$, an ϵ -rounded-up timed trace $[\xi]^{\epsilon}$ is a (possibly infinite) sequence $(t'_1, e_1)(t'_2, e_2)(t'_3, e_3) \dots$, such that there exists a sequence k_1, k_2, k_3, \dots , where $k_i \in \{0, \dots, \epsilon-1\}$, for all $i \ge 1$, $t'_i = [t_i] + k_i$, and t'_1, t'_2, t'_3, \dots is a non-decreasing sequence of timestamps.

The ϵ -rounded-up timed language $[L(\mathcal{A}_{\mathcal{T}\mathcal{A}})]^{\epsilon}$ over alphabet Σ is the set of all ϵ -rounded-up timed traces $[\xi]^{\epsilon}$ where $\xi \in L(\mathcal{A}_{T\mathcal{A}})$.

For $\epsilon = 1$, all k_i are equal to 0, and 1-rounded-up timed trace can be constructed just by rounding-up all timestamps of all timed events. Moreover, there is just one 1-rounded-up timed trace $\lceil \xi \rceil^1$ for each timed trace ξ . For example, for the timed trace $\xi = (0.5, a)(4.5, c)(7.6, b)(7.9, a) \dots$, the 1-rounded-up timed trace is $\lceil \xi \rceil^1 = (1, a)(5, c)(8, b)(8, a) \dots$

Intuitively, an event e_i occurring at a real-valued time point t_i should remain observable by the controller until the closest integral time point $\lceil t_i \rceil$. Therefore, all events with timestamp t_i , such that $\lceil t_i \rceil - \epsilon < t_i \le \lceil t_i \rceil$, are consumed by a digital machine at the nearest greater (or equal) integer point $\lceil t_i \rceil$.

For $\epsilon > 1$, there are several ϵ -rounded-up timed traces for each timed trace. Each ϵ -rounded-up timed trace describes at which (integer) time point each event is consumed. A timed event (t_i, e_i) can be consumed at any time point $t'_j = \lceil t_i \rceil + k$, where $k \in \{0, \ldots, \epsilon - 1\}$. Also, for each two timed events (t_1, e_1) and (t_2, e_2) in the same timed trace where $t_1 < t_2$ the event e_1 must be handled before the event e_2 . For example, for $\epsilon = 2$ and the timed trace ξ , we have two examples of digitalized timed traces: $\lceil \xi \rceil_1^2 = (2, a)(6, c)(9, b)(9, a) \dots$ and $\lceil \xi \rceil_2^2 = (1, a)(6, c)(8, b)(9, a) \dots$ In the following proposition we state the equivalence of ϵ -digitalized language and ϵ -rounded-up language. It will allow us to use ϵ -rounded-up timed traces instead of ϵ -digitalized ones and to exploit the fact that the former can be obtained from a timed trace in a natural way by rounding-up the timestamps of all its events.

Theorem 1. For each timed automaton \mathcal{A}_{TA} , $L^{\epsilon}(\mathcal{A}_{TA}) = [L(\mathcal{A}_{TA})]^{\epsilon}$.

Proof. By induction on the length of the timed trace accepted by A_{TA} .

3 Time-Triggered Automata

In this section, we present an automaton model for time-triggered systems and define the digitalized languages they accept. Time-triggered systems, in contrast to event-driven systems, run according to a given timetable. They have a global reference clock and the behavior of the system at each time point is determined by a part of the timetable corresponding to the value of the global clock. We want to remark again that time-triggered automata can be easily represented as timed automata and we introduce them only to capture the behavior of digital machines syntactically in a transparent way.

Similar to finite automata, time-triggered automata are finite labeled directed graphs. An input alphabet for these automata is a finite set of finite event sequences. We will refer to each such sequence as a *request*. We could use single events instead of sequences but this would make it awkward for the automata to be able to read several events at one integer time point. Each transition of a time-triggered automaton is labeled with a request and a time constraint from $\Theta = \{[n], (n) | n \in \mathbb{N}_{>0}\}$, where [n] (read as "at n") denotes instantaneous (non-periodic) constraints and (n) (read as "every n time units") denotes periodic constraints. Intuitively, constraints specify a pattern of time points at which requests are accepted by a time-triggered automaton. If a request is specified together with a periodic constraint then the automaton will check every n time units whether it is on the input. Instantaneous constraints determine only a single time point for handling the corresponding request.

Definition 6. A time-triggered automaton \mathcal{A}_{TTA} is a tuple $\langle \Sigma, S, s_0, T \rangle$ where

- $-\Sigma$ is a finite alphabet of events,
- -S is a finite set of locations,
- $-s_0 \in S$ is the initial location, and
- $-T \subseteq S \times \Sigma^* \times \Theta \times S$ is a finite set of edges.

We will use $s_1 \xrightarrow{\omega[n]} s_2$ to denote $(s_1, \omega, [n], s_2) \in T$ and $s_1 \xrightarrow{\omega(n)} s_2$ to denote $(s_1, \omega, (n), s_2) \in T$. We use symbol null to denote an empty sequence of events (sequence of length 0).

Figure 2 shows a time-triggered automaton recognizing the digitalized language of the timed automaton in Figure 1. For example, in state S_0 , every 1 time



Fig. 2. An example time-triggered automaton

unit, the automaton checks if a has occurred, and if a is observed, it consumes a and moves to S_1 . In S_1 , if ba is observed at time 3 (since it entered S_1), it consumes ba and moves back to S_1 . If nothing is observed (represented by *null*) in S_1 at time 3, the automaton moves to S_2 .

Time-triggered automata are non-deterministic in general. However, in order to make them implementable one has to deal with the deterministic subclass which can be obtained by prohibiting the following pairs of transitions outgoing from the same location:

$$\begin{array}{l} -s_1 \xrightarrow{\omega[n]} s_2, s_1 \xrightarrow{\omega[m]} s_3 \text{ where } n = m, \\ -s_1 \xrightarrow{\omega[n]} s_2, s_1 \xrightarrow{\omega(m)} s_3 \text{ where } m \mod n = 0, \text{ and} \\ -s_1 \xrightarrow{\omega(n)} s_2, s_1 \xrightarrow{\omega(m)} s_3. \end{array}$$

From now on we will consider only deterministic time-triggered automata.

3.1 Semantics

Generally, time-triggered automata can make the same types of transitions as timed automata, i.e. they can either change the location performing an action or delay in the same location. An automaton may change the location only when a transition outgoing from this location is enabled. Transitions are enabled at certain time points determined by their time constraints and the global time. Each edge with instantaneous constraint n is *enabled* when exactly n time units elapsed since the automaton has entered the location. Each edge with periodic constraint n is *enabled* every n time units after the automaton has entered the location.

A state of the time-triggered automaton A_{TTA} is a pair (s, t) where $s \in S$ is a location in A_{TTA} and $t \in \mathbb{R}_{\geq 0}$ is the time since A_{TTA} has entered s.

Definition 7. The semantics of a time-triggered automaton (S, Σ, s_0, T) with initial state $(s_0, 0)$ is a labeled transition system defined by the following rules:

$$\begin{array}{l} -(s_1,t) \xrightarrow{\omega} (s_2,0) \text{ if } s_1 \xrightarrow{\omega[n]} s_2 \text{ and } n=t \\ -(s_1,t) \xrightarrow{\omega} (s_2,0) \text{ if } s_1 \xrightarrow{\omega(m)} s_2 \text{ and } t \mod m=0, t>0 \\ -(s,t) \longrightarrow (s,t+d) \text{ if } t+d \leq \lfloor t+1 \rfloor \end{array}$$

The condition $t + d \leq \lfloor t + 1 \rfloor$, where $\lfloor n \rfloor$ is the greatest integer such that $\lfloor n \rfloor \leq n$, in the third rule ensures that no integer time point can be skipped by a delay transition. Allowing real-valued delays is not necessary here, but it will help us later when we will compose timed automata and time-triggered automata together. Note, that in the transition system for a deterministic \mathcal{A}_{TTA} there cannot be a state (s,t) with two outgoing transitions labeled by the same request ω . Now we can define a run of \mathcal{A}_{TTA} over a digital timed trace (time trace with integral timestamps only). Let $H(t,\omega) = (t,e_1)(t,e_2) \dots (t,e_n)$ where $\omega = e_1e_2 \dots e_n$ and $t \in \mathbb{N}_{>0}$ be the function unrolling a request associated with the timestamp t into a digital timed trace.

Definition 8. A run of a time-triggered automaton $\mathcal{A}_{TTA} = \langle S, \Sigma, s_0, T \rangle$ over a digital timed trace $[\xi] = (t_1, e_1)(t_2, e_2)(t_3, e_3) \dots$ is a sequence of the form

$$s_0 \xrightarrow[t_1]{\omega_1} s_1 \xrightarrow[t_2]{\omega_2} s_2 \xrightarrow[t_3]{\omega_3} \dots$$

where s_i is a location in A_{TTA} , satisfying the following requirements:

 $- H(t'_{1}, \omega_{1})H(t'_{2}, \omega_{2}) \cdots = (t_{1}, e_{1})(t_{2}, e_{2})(t_{3}, e_{3}) \cdots = [\xi]$ - for all $i \geq 1$ there is a transition $(s_{i-1}, t'_{i} - t'_{i-1}) \xrightarrow{\omega_{i}} (s_{i}, 0)$ and there is no transition $(s_{i-1}, t') \xrightarrow{\operatorname{null}} (s', 0), t' < t'_{i} - t'_{i-1}$ in the transition system induced by \mathcal{A}_{TTA} .

The language $L(A_{TTA})$ is the set of all digital timed traces $\lceil \xi \rceil$ for which there exists a run of A_{TTA} over $\lceil \xi \rceil$.

The first requirement says that \mathcal{A}_{TTA} must consume all timed events at the correct time and in the specified order. We consider consuming a request (a sequence of events) as consuming several events at the same time point, but in the order in which they appear in the request. The second requirement specifies when a request can be consumed. Note, that if only a transition labeled by *null* is enabled then it must be taken immediately. By this, we want to ensure deterministic behavior of \mathcal{A}_{TTA} .

4 Correctness Checking

We now show in which sense a time-triggered automaton \mathcal{A}_{TTA} handles correctly the events produced by a timed automaton \mathcal{A}_{TA} where each event expires within ϵ time units after having been released.

For time-triggered automata we apply maximal progress assumption, i.e. if a particular transition of A_{TTA} is enabled and there is a corresponding request

(sequence of events produced by A_{TA}) on the input it must be performed immediately. This gives us temporal determinism of the composition of A_{TA} and A_{TTA} . A_{TTA} does not have to guess whether to perform an action or whether to leave the events in the buffer and perform this action at the next time tick.

Ideally we want that for each timed trace of \mathcal{A}_{TA} , the corresponding ϵ -rounded-up timed trace should be accepted by \mathcal{A}_{TTA} . The problem is that when $\epsilon > 1$, there are several ϵ -rounded-up timed traces $[\xi]^{\epsilon}$ for each timed trace ξ . For example, the \mathcal{A}_{TA} shown in Figure 3(a) has a run over the timed trace $\xi = (0.5, a)(5, b)$. Given $\epsilon = 2$, there are several ϵ -rounded-up traces for ξ , for instance $[\xi]_1^2 = (1, a)(5, b)$ and $[\xi]_2^2 = (2, a)(5, b)$. Let us assume that we are given the \mathcal{A}_{TTA} shown in Figure 3(b) which can run over $[\xi]_2^2$ but cannot run over $[\xi]_1^2$. However, according to the maximal progress assumption, as *a* appears in ξ at 0.5, it should be picked up by \mathcal{A}_{TTA} no later than at 1. Unfortunately, \mathcal{A}_{TTA} does not accept $[\xi]_1^2$ which means that it is not correct with respect to \mathcal{A}_{TA} and ϵ (even if there exists an ϵ -rounded-up timed trace for ξ which is accepted by \mathcal{A}_{TTA}). Thus, for given ξ and ϵ we want to select just those ϵ -rounded-up timed traces which correspond to the maximal progress assumption for \mathcal{A}_{TTA} , namely $[\xi]_1^2$, and check whether they are accepted by \mathcal{A}_{TTA} .



Fig. 3. A timed automaton (a) and a time-triggered automaton (b)

Definition 9. Given \mathcal{A}_{TTA} and a finite timed trace ξ , we define the notion of \mathcal{A}_{TTA} promptly accepting an ϵ -rounded-up timed trace $[\xi]^{\epsilon}$ inductively.

- 1. If $|\xi|=0$ ($|\xi|$ is the length of ξ) then $[\xi]^{\epsilon}$ is accepted promptly.
- 2. If $\xi = \xi' \cdot (t, e)$ then $[\xi]^{\epsilon} = [\xi']^{\epsilon} \cdot (t', e)$ is accepted promptly iff $[\xi']^{\epsilon}$ is accepted promptly, $[\xi]^{\epsilon}$ is accepted by \mathcal{A}_{TTA} , and no ϵ -rounded-up timed trace $[\xi']^{\epsilon} \cdot (t'', e)$ for ξ where t'' < t' is also accepted by \mathcal{A}_{TTA} .

Finally we say that (possibly infinite) sequence $[\xi]^{\epsilon}$ is accepted promptly iff all its finite prefixes are accepted promptly.

Now we can define our notion of correctness.

Definition 10. Given $\epsilon \in \mathbb{N}_{>0}$, a timed automaton \mathcal{A}_{TA} , and a time-triggered automaton \mathcal{A}_{TTA} , we say that \mathcal{A}_{TTA} is correct with respect to \mathcal{A}_{TA} and ϵ iff for each timed trace $\xi \in L(\mathcal{A}_{TA})$ there exists an ϵ -rounded-up timed trace $[\xi]^{\epsilon}$ which is promptly accepted by \mathcal{A}_{TTA} .

As ϵ -rounded-up timed trace specifies all servicing time points within the deadline and prompt acceptance corresponds to a possible run of \mathcal{A}_{TTA} (satisfying the maximal progress assumption) in which all events are picked-up in specified time points, the above definition ensures that no deadline will be missed.

For the case $\epsilon = 1$, it follows at once that the correctness criterion can be captured by a simple language inclusion property as stated below.

Proposition 1. For $\epsilon = 1$, a time-triggered automaton \mathcal{A}_{TTA} is correct with respect to a timed automaton \mathcal{A}_{TA} and ϵ iff $[L(\mathcal{A}_{TA})]^{\epsilon} \subseteq L(\mathcal{A}_{TTA})$.

Proof. For $\epsilon = 1$, prompt acceptance coincides with (simple) acceptance and for each timed trace ξ there exists exactly one ϵ -rounded-up timed trace $\lceil \xi \rceil^{\epsilon}$. According to Definition 5, $\lceil L(A_{TA}) \rceil^{\epsilon} = \{\lceil \xi \rceil^{\epsilon} \mid \xi \in L(A_{TA})\}$.

We now propose an effective method (which can in fact be automated) to verify the correctness of A_{TTA} with respect to A_{TA} and ϵ . Our strategy is to reduce this verification problem to the problem of checking schedulability for timed automata extended with asynchronous tasks.

In [FPY02] a classical notion of schedulability has been extended to timed automata asynchronously releasing tasks for execution when discrete transitions are taken. Tasks have to complete their execution before the deadline. A processing unit (e.g. CPU) executes the released tasks stored in a queue according to some scheduling policy (e.g. FPS or EDF). The goal of the schedulability analysis is then to check that all the released tasks meet their deadlines along all possible runs of an automaton.

Since we consider \mathcal{A}_{TTA} as a device which handles requests that come from \mathcal{A}_{TA} , it is natural to interpret \mathcal{A}_{TA} as a timed automaton extended with asynchronous tasks. Each request produced by \mathcal{A}_{TA} corresponds to a task annotated with the relative deadline equal to ϵ . When released by \mathcal{A}_{TA} , a task is put in the task queue, which we will denote Ω . In its turn, \mathcal{A}_{TTA} can be seen as a processing unit (Run-function in [FPY02]) that executes the ready queue of tasks according to a scheduling strategy (First Come First Served (FIFO), in our case). The setting used to perform an automatic verification of the correctness of \mathcal{A}_{TTA} with respect to \mathcal{A}_{TA} and ϵ is shown in Figure 4.



Fig. 4. The setting for verification of correctness of A_{TTA} w.r.t. A_{TA} , ϵ

We define a machine $\mathbb{M}_{\mathcal{A}_{TA}, \epsilon, \mathcal{A}_{TTA}}$ used for automatic correctness checking of \mathcal{A}_{TTA} with respect to \mathcal{A}_{TA} and ϵ as a triple $\langle \mathcal{A}_{TA}, \epsilon, \mathcal{A}_{TTA} \rangle$. A state of $\mathbb{M}_{\mathcal{A}_{TA},\epsilon,\mathcal{A}_{TTA}}$ is (l, u, q, s, t) where l is a location in \mathcal{A}_{TA} , u is a clock assignment, q is a state of the task queue Ω , s is a location in \mathcal{A}_{TTA} , and $t \in \mathbb{R}_{\geq 0}$ is the time elapsed since A_{TTA} has entered s.

We denote q :: e the queue with a pair (e, ϵ) inserted in the back of the queue, and $q \mid \omega, \omega = e_1 e_2 \dots e_n$ the queue with $(e_1, \delta_1), (e_2, \delta_2), \dots, (e_n, \delta_n)$ removed from it, where $(e_1, \delta_1), (e_2, \delta_2), \dots, (e_n, \delta_n)$ are the *n* foremost pairs in the queue $(\text{Head}(q) = (e_1, \delta_1))$. Otherwise, it is not possible to perform $q \setminus \omega$. We remove events from the queue in the order they have been inserted (FIFO).

Definition 11. The semantics of $\mathbb{M}_{\mathcal{A}_{TA},\epsilon,\mathcal{A}_{TTA}}$, where $\mathcal{A}_{TA} = \langle \mathcal{C}, \Sigma, N, l_0, E \rangle$, $\mathcal{A}_{TTA} = \langle \Sigma, S, s_0, T \rangle$, with initial state $(l_0, u_0, q_0, s_0, 0)$ is a labeled transition system defined by the following rules:

- $\begin{array}{l} -(l,u,q,s,t) \stackrel{e}{\longrightarrow} (l',u[r\mapsto 0],q::e,s,t), \ if \ (l,\phi,e,r,l') \in E, \ and \ u \models \phi, \\ -(l,u,q,s,t) \longrightarrow (l,u,q \backslash \omega, s_1, 0), \ if \ t \in \mathbb{N}_{>0}, \ it \ is \ possible \ to \ perform \ q \backslash \omega, \end{array}$ and $s \xrightarrow{\boldsymbol{\omega}[t]} s_1$, or $s \xrightarrow{\boldsymbol{\omega}(n)} s_1$ where $t \mod n = 0$,
- $(l, u, q, s, t) \xrightarrow{d} (l, u + d, q d, s, t + d)$, if $t + d \leq \lfloor t + 1 \rfloor$, and if $t \in \mathbb{N}_{>0}$ then there is no enabled edge labeled with request ω in \mathcal{A}_{TTA} such that it is possible to perform $q \setminus \omega$.

A state (l, u, q, s, t) of $\mathbb{M}_{\mathcal{A}_{TA}, \epsilon, \mathcal{A}_{TTA}}$ is called *faulty* if for some pair $(e, \delta) \in q$, an event e misses its deadline, i.e. $\delta \leq 0$. $\mathbb{M}_{\mathcal{A}_{TA},\epsilon,\mathcal{A}_{TTA}}$ is called *schedulable* iff no faulty state is reachable from the initial state of $\mathbb{M}_{\mathcal{A}_{TA},\epsilon,\mathcal{A}_{TTA}}$.

Theorem 2. A_{TTA} is correct with respect to A_{TA} and ϵ iff $\mathbb{M}_{A_{TA},\epsilon,A_{TTA}}$ is schedulable. Further, one can effectively decide the schedulability of $\mathbb{M}_{\mathcal{A}_{TA},\epsilon,\mathcal{A}_{TTA}}$.

Sketch of Proof. The first part of the result follows from the construction of $\mathbb{M}_{\mathcal{A}_{TA},\epsilon,\mathcal{A}_{TTA}}$. According to Definition 10 we are looking for ϵ -rounded-up timed traces promptly accepted by A_{TTA} for each run of A_{TA} . As follows from Proposition 1 and Definition 3, each such ϵ -rounded-up timed trace corresponds to a path in the labeled transition system \mathcal{S}^{ϵ} for \mathcal{A}_{TA} . We can view prompt acceptance of such a word as a path in the transition system obtained as the synchronous product of \mathcal{S}^{ϵ} and the transition system induced by \mathcal{A}_{TTA} . Now, production, consumption, and delay transitions in this product correspond to the transitions of $\mathbb{M}_{\mathcal{A}_{TA},\epsilon,\mathcal{A}_{TTA}}$, respectively. Faulty states in $\mathbb{M}_{\mathcal{A}_{TA},\epsilon,\mathcal{A}_{TTA}}$ correspond to states in S^{ϵ} where event consumption failed.

The second part of the result boils down to a straightforward adaptation of the decidability argument for the schedulability of timed automata extended with tasks developed in [FPY02]. In this work, a timed automaton extended with tasks, the task queue, the Run-function, and a scheduling policy are encoded as a timed automaton and the schedulability problem is reduced to the reachability of a predefined faulty state. As A_{TTA} corresponding to the Run-function is deterministic, it can be easily encoded in this timed automaton instead of the Run-function.

5 Conclusions

In this paper we propose to use timed automata (TAs) as event-triggered models for real time applications. A real time application may be a plant or simply a real time environment that generates sequences of events according to the pattern and timing constraints specified by a timed automaton. The sequences of events are picked up and served by a digital controller. We are aiming at synthesizing executable code (implementing a digital controller) for time-triggered architectures from such event-triggered specifications. As a first step, we abstract away from the computation tasks associated with the events, and only focus on the mechanism for admitting the events into the time-triggered environment before their observability deadlines expire. We have developed an automaton model called time-triggered automata (TTAs) to model finite state implementations of a controller that services the request patterns modeled by a timed automaton. A time-triggered automaton is deterministically driven by a timetable defined w.r.t. a fixed granularity of time. It is intended to be a finite state abstraction of computations realized on a time-triggered architectures [KB01,Kop98].

To relate the behaviors specified using TAs and TTAs, we have proposed a new semantics for TAs based on the non-instant observability of events, which gives rise to a simple notion of digitalization of timed languages. This enables to formulate the correctness criterion on TTA implementations of TA specifications as a language inclusion property. Our main result is that we can effectively decide whether a TTA correctly services the request patterns generated by a TA, that is, the TTA implements the TA specification.

We hope that the results presented in this paper may serve as a semantic basis for automatic code generation from abstract specifications in real time settings. Currently we are looking at how to automatically synthesize time-triggered automata from timed automata according to the correctness criterion presented in this paper.

References

- [AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
- [AFP⁺02] T. Amnell, E. Fersman, P. Pettersson, H. Sun, and W. Yi. Code synthesis for timed automata. *Nordic Journal of Computing*, 9(4):269–300, 2002.
- [BCP+01] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys = Esterel + Kronos. A tool for verifying real-time properties of embedded systems. In *Proceedings of International Conference* on Decision and Control, CDC'01. IEEE Control Systems Society, 2001.
- [BDM⁺98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In *Proceedings of CAV'98*, volume 1427 of *LNCS*, pages 546–550. Springer–Verlag, 1998.
- [BGK⁺96] J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In *Proceedings of CAV'96*, pages 244–256. LNCS, 1996.

- [BPS00] Valérie Bertin, Michel Poize, and Joseph Sifakis. Towards validated realtime software. In Proceedings of the 12 th Euromicro Conference on Real Time Systems, pages 157–164. IEEE, 2000.
- [CPP+01] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys: a tool for the development and verification real-time embedded systems. In *Proceedings of CAV'01*, volume 2102 of *LNCS*. Springer–Verlag, 2001.
- [DY00] A. David and W. Yi. Modeling and analysis of a commercial field bus protocol. In *Proceedings of Euromicro Conference on Real-Time*. IEEE Computer Society, 2000.
- [FPY02] E. Fersman, P. Pettersson, and W. Yi. Timed automata with asynchronous processes: Schedulability and decidability. In *Proceedings of TACAS'02*, volume 2280 of *LNCS*, pages 67–82. Springer–Verlag, 2002.
- [GHJ97] V. Gupta, T. Henzinger, and R. Jagadeesan. Robust timed automata. In Hybrid and Real-Time Systems, pages 331–345, Grenoble, France, 1997. Springer Verlag, LNCS 1201.
- [HKH03] T.A. Henzinger, C.M. Kirsch, and B. Horowitz. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, January 2003.
- [HMP92] T. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Proceedings of ICALP'92*, volume 623 of *LNCS*, pages 545–558. Springer-Verlag, 1992.
- [KB01] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings* of the IEEE, Special Issue on Modeling and Design of Embedded Software, 2001.
- [Kop98] H. Kopetz. The time-triggered model of computation. *Proceedings of the* 19th IEEE Systems Symposium (RTSS'98), 1998.
- [KY04] Pavel Krčál and Wang Yi. Decidable and undecidable problems in schedulability analysis using timed automata. In *Proceedings of TACAS'04*, volume 2988 of *LNCS*, pages 236–250. Springer–Verlag, 2004.
- [LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. Int. Journal on Software Tools for Technology Transfer, 1(1–2):134–152, October 1997.
- [LY97] K. G. Larsen and W. Yi. Time-abstracted bisimulation: Implicit specifications and decidability. *Information and Computation*, 134(2):75–101, 1997.
- [OW03] J. Ouaknine and J. Worrell. Revisiting digitization, robustness and decidability for timed automata. In *Proceedings of LICS'03*, pages 198–207. IEEE Press, 2003.
- [STY03] J. Sifakis, S. Tripakis, and S. Yovine. Building models of real-time systems from application software. *Proceedings of the IEEE, Special issue on modeling and design of embedded systems*, 91(1):100–111, 2003.
- [WDR04] M. De Wulf, L. Doyen, and J.-F. Raskin. Almost ASAP semantics: From timed models to timed implementations. In *Proceedings of HSCC'04*, volume 2993 of *Lecture Notes in Computer Science*, pages 296–310. Springer-Verlag, 2004.
- [WH04] L. Waszniowski and Z. Hanzálek. Analysis of real time operating system based applications. In *Proceedings of FORMATS'03*, volume 2791 of *LNCS*, pages 219–233. Springer-Verlag, 2004.

Extended Process Rewrite Systems: Expressiveness and Reachability*

Mojmír Křetínský, Vojtěch Řehák**, and Jan Strejček

Faculty of Informatics, Masaryk University Brno Botanická 68a, 602 00 Brno, Czech Republic {kretinsky, rehak, strejcek}@fi.muni.cz

Abstract. We unify a view on three extensions of Process Rewrite Systems (PRS) and compare their expressive power with that of PRS. We show that the class of Petri nets is less expressive up to bisimulation equivalence than the class of PA processes extended with a finite state control unit. Further we show our main result that the reachability problem for PRS extended with a so called weak finite state unit is decidable.

1 Introduction

An automatic verification of current software systems often needs to model them as infinite-state systems, i.e. systems with an evolving structure and/or operating on unbounded data types. Infinite-state systems can be specified in a number of ways with their respective advantages and limitations. Petri nets, pushdown automata, and process algebras like BPA, BPP, or PA all serve to exemplify this. Here we employ the classes of infinite-state systems defined by term rewrite systems and called *Process Rewrite Systems* (PRS) as introduced by Mayr [May00]. PRS subsume a variety of the formalisms studied in the context of formal verification (e.g. all the models mentioned above).

A PRS is a finite set of rules $t \xrightarrow{a} t'$ where *a* is an action under which a subterm *t* can be reduced onto a subterm *t'*. Terms are built up from an empty process ε and a set of process constants using (associative) sequential "." and (associative and commutative) parallel "||" operators. The semantics of PRS can be defined by labelled transition systems (LTS) – labelled directed graphs whose nodes (states of the system) correspond to terms modulo properties of "." and "||" and edges correspond to individual actions (computational steps) which can be performed in a given state. The relevance of various subclasses of PRS for modelling and analysing programs is shown e.g. in [Esp02], for automatic verification see e.g. surveys [BCMS01,Srb02].

^{*} This work has been supported by GAČR, grant No. 201/03/1161.

^{**} The co-author has been supported by Marie Curie Fellowship of the European Community Programme Improving the Human Research Potential and the Socioeconomic Knowledge Base under contract number HPMT-CT-2000-00093.

Mayr [May00] has also shown that the reachability problem (i.e. given terms t, t': is t reducible to t'?) for PRS is decidable. Most research (with some recent exceptions, e.g. [BT03,Esp02]) has been devoted to the PRS classes from the lower part of the PRS hierarchy, especially to pushdown automata (PDA), Petri nets (PN) and their respective subclasses. We mention the successes of PDA in modeling recursive programs (without process creation), PN in modeling dynamic creation of concurrent processes (without recursive calls), and CPDS (communicating pushdown systems [BET03]) modeling both features. All of these formalisms subsume a notion of a finite state unit (FSU) keeping some kind of global information which is accessible to the redices (the ready to be reduced components) of a PRS term – hence a FSU can regulate rewriting. On the other hand, using a FSU to extend the PRS rewriting mechanism is very powerful since the state-extended version of PA processes (sePA) has a full Turing-power [BEH95] – the decidability of reachability is lost for sePA, including all its superclasses (see Figure 1), and CPDS as well.

This paper presents a hierarchy of PRS classes and their respective extensions of three types: fcPRS classes ([Str02], inspired by concurrent constraint programming [SR90]), wPRS classes ([KŘS03], PRS systems equipped with weak FSU inspired by weak automata [MSS92]), and state-extended PRS classes [JKM01]. The classes in the hierarchy (depicted in Figure 1) are related by their expressive power with respect to (strong) bisimulation equivalence. As the main contribution of the paper we show that the reachability problem remains decidable for the very expressive class of wPRS. This result deserves some additional remarks:

- It determines the decidability borderline of the reachability problem in the mentioned hierarchy; the problem is decidable for all classes except those with Turing power. In other words, it can be seen as a contribution to studies of algorithmic boundaries of reachability for infinite-state systems.
- In the context of verification, one often formulates a property expressing that *nothing bad occurs*. These properties are called *safety properties*. The collection of the most often verified properties [DAC98] contains 41% of such properties. Model checking of safety properties can be reduced to the reachability problem. Moreover, many successful verification tools concentrate on reachability only. Therefore, our decidability result can be seen as a contribution to an automatic verification of infinite-state systems as well.
- Given a labelled transition system $(S, Act, \rightarrow, \alpha_0)$ with a distinguished action $\tau \in Act$, we define a *weak trace set* of a state $s \in S$ as

$$wtr(s) = \{w \in (Act \smallsetminus \{\tau\})^* \mid s \stackrel{w}{\Longrightarrow} t \text{ for some } t \in S\},\$$

where $s \stackrel{w}{\Longrightarrow} t$ means that there is some $w' \in Act^*$ such that $s \stackrel{w'}{\longrightarrow} t$ and w is equal to w' without τ actions. Two systems are *weak trace equivalent* if the weak trace sets of their initial states are the same. So far it has been known that weak trace non-equivalence is semi-decidable for Petri nets (see e.g. [Jan95]), pushdown processes (due to [Büc64]), and PA processes (due to [LS98]). Using the decidability result, it is easy to show that the weak

trace set is recursive for every state of any wPRS. Hence, the weak trace non-equivalence is semi-decidable for (all subclasses of) wPRS.

– Our decidability result has been recently applied in the area of cryptographic protocols. Hüttel and Srba [HS04] define a replicative variant of a calculus for Dolev and Yao's ping-pong protocols [DY83]. They show that the reachability problem for these protocols is decidable as it can be reduced to the reachability problem for wPRS.

The outline of the paper is as follows: after some preliminaries we introduce a uniform framework for specifying all extended PRS formalisms in Section 3 and compare their relative expressiveness with respect to bisimulation equivalence in Section 4. Here we also solve (to the best of our knowledge) an open problem on the relationship between the PN and sePA classes by showing that PN is less expressive (up to bisimulation equivalence) than sePA. In Section 5 we show that all classes of our fcPRS and wPRS extensions keep the reachability problem decidable. The last section summarises our results.

Related Work: In the context of reachability analysis one can see at least two approaches: (i) abstraction (approximate) analysis techniques on stronger 'models' such as sePA and its superclasses with undecidable reachability, e.g. see a recent work [BET03], and (ii) precise techniques for 'weaker' models, e.g. PRS classes with decidable reachability, e.g. [LS98] and another recent work [BT03]. In the latter one, symbolic representations of set of reachable states are built with respect to various term structural equivalences. Among others it is shown that for the PAD class and the same equivalence as in this paper, when properties of sequential and parallel compositions are taken into account, one can construct nonregular representations based on counter tree automata.

2 Preliminaries

A labelled transition system (LTS) \mathcal{L} is a tuple $(S, Act, \rightarrow, \alpha_0)$, where S is a set of states or processes, Act is a set of atomic actions or labels, $\rightarrow \subseteq S \times Act \times S$ is a transition relation (written $\alpha \xrightarrow{a} \beta$ instead of $(\alpha, a, \beta) \in \rightarrow$), $\alpha_0 \in S$ is a distinguished *initial state*.

We use the natural generalization $\alpha \xrightarrow{\sigma} \beta$ for finite sequences of actions $\sigma \in Act^*$. The state α is *reachable* if there is $\sigma \in Act^*$ such that $\alpha_0 \xrightarrow{\sigma} \alpha$.

A binary relation *R* on set of states *S* is a *bisimulation* [Mil89] iff for each $(\alpha, \beta) \in R$ the following conditions hold:

$$\begin{array}{l} - \ \forall \ \alpha' \in S, a \in Act : \alpha \xrightarrow{a} \alpha' \Longrightarrow (\exists \beta' \in S : \beta \xrightarrow{a} \beta' \land (\alpha', \beta') \in R) \\ - \ \forall \ \beta' \in S, a \in Act : \beta \xrightarrow{a} \beta' \Longrightarrow (\exists \alpha' \in S : \alpha \xrightarrow{a} \alpha' \land (\alpha', \beta') \in R) \end{array}$$

Bisimulation equivalence (or *bisimilarity*) on a LTS is the union of all bisimulations (i.e. the largest bisimulation).

Let $Const = \{X, ...\}$ be a countably infinite set of *process constants*. The set \mathcal{T} of *process terms* (ranged over by t, ...) is defined by the abstract syntax

 $t = \varepsilon \mid X \mid t_1.t_2 \mid t_1 \mid t_2$, where ε is the empty term, $X \in Const$ is a process constant (used as an atomic process), ' \mid ' and '.' mean parallel and sequential compositions respectively.

The set Const(t) is the set of all constants occurring in a process term t. We always work with equivalence classes of terms modulo commutativity and associativity of '||' and modulo associativity of '.' We also define $\varepsilon t = t = t.\varepsilon$ and $t ||\varepsilon = t$.

We distinguish four classes of process terms as:

- 1 terms consisting of a single process constant only, in particular $\varepsilon \notin 1$,
- S-sequential terms without parallel composition, e.g. X.Y.Z,
- P parallel terms without sequential composition, e.g. X ||Y||Z,

G – general terms with arbitrarily nested sequential and parallel compositions.

Definition 1. Let $Act = \{a, b, \dots\}$ be a countably infinite set of atomic actions, $\alpha, \beta \in \{1, S, P, G\}$ such that $\alpha \subseteq \beta$. An (α, β) -PRS (process rewrite system) Δ is a pair (R, t_0) , where

- *R* is a finite set of rewrite rules of the form $t_1 \xrightarrow{a} t_2$, where $t_1 \in \alpha$, $t_1 \neq \varepsilon$, $t_2 \in \beta$ are process terms and $a \in Act$ is an atomic action,
- $-t_0 \in \beta$ is an initial state.

Given PRS Δ we define $Const(\Delta)$ as the set of all constants occurring in the rewrite rules of Δ or in its initial state, and $Act(\Delta)$ as the set of all actions occurring in the rewrite rules of Δ . We sometimes write $(t_1 \xrightarrow{a} t_2) \in \Delta$ instead of $(t_1 \xrightarrow{a} t_2) \in R$.

The semantics of Δ is given by the LTS $(S, Act(\Delta), \rightarrow, t_0)$, where $S = \{t \in \beta \mid Const(t) \subseteq Const(\Delta)\}$ and \rightarrow is the least relation satisfying the inference rules:

$$\frac{(t_1 \xrightarrow{a} t_2) \in \Delta}{t_1 \xrightarrow{a} t_2}, \qquad \frac{t_1 \xrightarrow{a} t_1'}{t_1 \| t_2 \xrightarrow{a} t_1' \| t_2}, \qquad \frac{t_1 \xrightarrow{a} t_1'}{t_1 . t_2 \xrightarrow{a} t_1' . t_2}.$$

If no confusion arises, we sometimes speak about a "process rewrite system" meaning a "labelled transition system generated by process rewrite system".

Some classes of (α, β) -PRS correspond to widely known models as finite state systems (FS), basic process algebras (BPA), basic parallel processes (BPP), process algebras (PA), pushdown processes (PDA, see [Cau92] for justification), and Petri nets (PN). The other classes were introduced (and named as PAD, PAN, and PRS) by Mayr [May00]. The correspondence between (α, β) -PRS classes and acronyms just mentioned can be seen in Figure 1.

3 Extended PRS

In this section we recall the definitions of three different extensions of process rewrite systems, namely *state-extended PRS (sePRS)* [JKM01], *PRS with a finite constraint system (fcPRS)* [Str02], and *PRS with a weak finite-state unit* (wPRS) [KŘS03]. In all cases, the PRS formalism is extended with a finite state unit of some kind.

sePRS. State-extended PRS corresponds to PRS extended with a finite state unit without any other restrictions. The well-known example of this extension is the state-extended BPA class (also known as pushdown processes).

wPRS. The notion of weakness employed in the wPRS formalism corresponds to that of weak automaton [MSS92] in automata theory. The behaviour of a weak state unit is acyclic, i.e. states of state unit are ordered and non-increasing during every sequence of actions. As the state unit is finite, its state can be changed only finitely many times during every sequence of actions.

fcPRS. The extension of PRS with finite constraint systems is motivated by *concurrent constraint programming (CCP)* (see e.g. [SR90]). In CCP the processes work with a shared *store* (seen as a constraint on values that variables can represent) via two operations, *tell* and *ask*. The *tell* adds a constraint to the store provided the store remains *consistent*. The *ask* is a test on the store – it can be executed only if the current store implies a specified constraint.

Formally, values of a store form a bounded lattice (called a *constraint system*) with the lub operation \land (least upper bound), the least element *tt*, and the greatest element *ff*. The execution of tell(n) changes the value of the store from o to $o \land n$ (provided $o \land n \neq ff$ - consistency check). The ask(m) can be executed if the current value of the store o is greater than m.

The state unit of fcPRS has the same properties as the store in CCP. We add two constraints (m, n) to each rewrite rule. The application of a rule corresponds to the concurrent execution of ask(m), tell(n), and rewriting:

- a rule can be applied only if the actual store o satisfies $m \leq o$ and $o \land n \neq ff$,
- the application of the rule rewrites the process term and changes the store to $o \wedge n$.

We first define the common syntax of the aforementioned extended PRS and then we specify the individual restrictions on state units.

Definition 2. Let $Act = \{a, b, \dots\}$ be a countably infinite set of atomic actions, $\alpha, \beta \in \{1, S, P, G\}$ such that $\alpha \subseteq \beta$. An extended (α, β) -PRS Δ is a tuple (M, \leq, R, m_0, t_0) , where

- M is a finite set of states of the state unit,
- \leq is a binary relation over M,
- $\overline{R} \text{ is a finite set of rewrite rules of the form } (m, t_1) \xrightarrow{a} (n, t_2), \text{ where } t_1 \in \alpha, \\ t_1 \neq \varepsilon, t_2 \in \beta, m, n \in M, \text{ and } a \in Act,$
- Pair $(m_0, t_0) \in M \times \beta$ forms a distinguished initial state of the system.

The specific type of an extended (α, β) -PRS is given by further requirements on \leq . An extended (α, β) -PRS is

- $-(\alpha,\beta)$ -sePRS without any requirements on \leq .¹
- $-(\alpha,\beta)$ -wPRS iff (M,\leq) is a partially ordered set.
- $-(\alpha, \beta)$ -*fcPRS* iff (M, \leq) is a bounded lattice. The lub operation (least upper bound) is denoted by \wedge , the least and the greatest elements are denoted by *tt* and *ff*, respectively. We also assume that $m_0 \neq ff$.

To shorten our notation we prefer *mt* over (m, t). As in the PRS case, instead of $(mt_1 \xrightarrow{a} nt_2) \in R$ where $\Delta = (M, \leq, R, m_0, t_0)$, we usually write $(mt_1 \xrightarrow{a} nt_2) \in \Delta$. The meaning of $Const(\Delta)$ (process constants used in rewrite rules or in t_0) and $Act(\Delta)$ (actions occurring in rewrite rules) for a given extended PRS Δ is also the same as in the PRS case.

The semantics of an extended (α, β) -PRS system Δ is given by the corresponding labelled transition system $(S, Act(\Delta), \rightarrow, m_0t_0)$, where²

$$S = M \times \{t \in \beta \mid Const(t) \subseteq Const(\Delta)\}$$

and the relation \longrightarrow is defined as the least relation satisfying the inference rules corresponding to the application of rewrite rules (and dependent on the concrete formalism):

sePRS $\frac{(mt_1 \xrightarrow{a} nt_2) \in \Delta}{mt_1 \xrightarrow{a} nt_2}$ wPRS $\frac{(mt_1 \xrightarrow{a} nt_2) \in \Delta}{mt_1 \xrightarrow{a} nt_2}$ if $n \le m$ fcPRS $\frac{(mt_1 \xrightarrow{a} nt_2) \in \Delta}{ot_1 \xrightarrow{a} (o \land n)t_2}$ if $m \le o$ and $o \land n \neq ff$

and two common inference rules

$$\frac{mt_1 \xrightarrow{a} nt'_1}{m(t_1||t_2) \xrightarrow{a} n(t'_1||t_2)}, \qquad \frac{mt_1 \xrightarrow{a} nt'_1}{m(t_1.t_2) \xrightarrow{a} n(t'_1.t_2)},$$

where $t_1, t_2, t'_1 \in \mathcal{T}$ and $m, n, o \in M$.

Instead of (1,*S*)-sePRS, (1,*S*)-wPRS, (1, *S*)-fcPRS,... we use a more natural notation seBPA, wBPA, fcBPA, etc. The class seBPP is also known as *multiset automata* (*MSA*) or *parallel pushdown automata* (*PPDA*), see [Mol96].

4 Expressiveness

Figure 1 describes the hierarchy of PRS classes and their extended counterparts with respect to bisimulation equivalence. If any process in class *X* can be also defined (up to bisimilarity) in class *Y* we write $X \subseteq Y$. If additionally $Y \not\subseteq X$ holds, we write $X \subsetneq Y$ and say *X* is less expressive than *Y*. This is depicted by

¹ In this case, the relation \leq can be omitted from the definition.

² If Δ is an fcPRS, we eliminate the states with *ff* from *S* as they are unreachable.



Fig. 1. The hierarchy of classes defined by (extended) rewrite formalisms

the line(s) connecting X and Y with Y placed higher than X in Figure 1. The dotted lines represent the facts $X \subseteq Y$, where we conjecture that $X \subsetneq Y$ hold.

Some observations (even up to isomorphism) are immediate, for example

- 1. the classes FS, PDA and PN coincide with their extended analogues,
- 2. if $e \in \{se, w, fc\}$ and $X \subseteq Y$ then $eX \subseteq eY$, and
- 3. (α, β) -PRS $\subseteq (\alpha, \beta)$ -fcPRS $\subseteq (\alpha, \beta)$ -wPRS $\subseteq (\alpha, \beta)$ -sePRS for all (α, β) -PRS.

The strictness (' \subsetneq ') of the PRS-hierarchy has been proved by Mayr [May00], that of the corresponding classes of PRS and fcPRS has been proved in [Str02], and the relations among MSA and the classes of fcPRS and wPRS have been studied in [KŘS03]. Note that the strictness relations wX \subsetneq seX hold for all X = PA, PAD, PAN, PRS due to our reachability result for wPRS given in Sec. 5 and due to the full Turing-power of sePA [BEH95].

These proofs together with Moller's result establishing MSA \subsetneq PN [Mol98] complete the justification of Figure 1 – with one exception, namely the relation between the PN and sePA classes. Looking at two lines leaving sePA down to the left and down to the right, we note the "left-part collapse" of (*S*, *S*)-PRS and PDA proved by Caucal [Cau92] (up to isomorphism). The right-part counterpart is slightly different due to the previously mentioned result that MSA \subsetneq PN. In the next subsection we prove that PN \subsetneq sePA (in fact it suffices to demonstrate PN \subseteq sePA as the strictness is obvious).

4.1 $PN \subsetneq sePA$

We now show that Petri nets are less expressive (with respect to bisimilarity) than sePA processes. In this section, a Petri net Δ is considered in traditional notation (see e.g. [Pet81]). Let $Const(\Delta) = \{X_1, \ldots, X_k\}$, a state $X_1^{p_1} \| \ldots \| X_k^{p_k}$ of a PN Δ is written as (p_1, \ldots, p_k) and called *marking*. Each p_i is the number of tokens at the place P_i . Any rewrite rule $X_1^{l_1} \| \ldots \| X_k^{l_k} \xrightarrow{a} X_1^{r_1} \| \ldots \| X_k^{r_k}$ (where $l_i, r_i \geq 0$) is written as $(l_1, \ldots, l_k) \xrightarrow{a} (r_1, \ldots, r_k)$ and called *transition*³. The heart of our argument is a construction of a sePA Δ' bisimilar to a given PN Δ .

The main difficulty in this construction is to maintain the number of tokens at the places of a PN. To this end, we may use two types of sePA memory: a finite control (FSU), which cannot represent an unbounded counter, and a term of an unbounded length, where just one constant can be rewritten in one step.

Our construction of a sePA Δ' can be reformulated on intuitive level as follows. Let a marking (p_1, \ldots, p_k) mean that we have p_i units of the *i*-th currency, $i = 1, \ldots, k$. An application of a PN transition $(l_1, \ldots, l_k) \stackrel{a}{\longrightarrow} (r_1, \ldots, r_k)$ has the effect of a currency exchange from p_i to $p_i - l_i + r_i$ for all *i*. A sePA reseller Δ' will have *k* finite pockets (in its FSU) and *k* bank accounts (a parallel composition of *k* sequential terms t_i). The reseller Δ' maintains an invariant $p_i = pocket_i + account_i$ for all *i*. To mimic a PN transition he must obey sePA rules, i.e. he may use all his pockets, but just one of his accounts in one exchange. A solution is to do $pocket_i \leftrightarrow account_i$ transfers cyclically, $i = 1, \ldots, k$. Hence, rebalancing $pocket_i$ the reseller Δ' must be able to perform the next k-1 exchanges without accessing $account_i$ (while visiting the other accounts). Therefore, Δ' needs sufficiently large (but finite) pockets and sufficiently high (and fixed) limits for $pocket_i \leftrightarrow account_i$ transfers. We show these bounds exist.

In one step the amount of the *i*-th currency cannot be changed by more than $L_i = max \{ |l_i - r_i|; (l_1, \ldots, l_k) \xrightarrow{a} (r_1, \ldots, r_k) \text{ is a PN transition} \}$, thus $M_i = k \cdot L_i$ is an upper bound for the total effect of k consecutive steps. Any rebalancing of *pocket_i* sets its value into $\{M_i, \ldots, 2M_i - 1\}$ (or $\{0, \ldots, 2M_i - 1\}$ if *account_i* is empty). Hence, after k transitions the value of *pocket_i* is in $\{0, \ldots, 3M_i - 1\}$. In the next rebalancing *account_i* can be increased or decreased (if it is not empty) by M_i to get *pocket_i* between M_i (or 0 if *account_i* is empty) and $2M_i - 1$ again.

³ Till now, X^n denoted the parallel composition of *n* copies of *X*. In the rest of the section, it denotes the sequential composition of *n* copies of *X*.

Each state of sePA Δ' consists of a state of a FSU and a term (parallel composition of k stacks representing accounts). A state of a FSU is in the product

$$\{1, \ldots, k\} \times \{0, \ldots, 3M_1 - 1\} \times \ldots \times \{0, \ldots, 3M_k - 1\}.$$

$$update \ controller \qquad pocket_1 \qquad pocket_k$$

The *update controller* goes around the range and refers to the account being updated (rebalanced) in the next step. The value of each $pocket_i$ (subsequently denoted by m_i) is equal to the number of tokens at P_i counted modulo M_i .

We define 2k process constants $B_i, X_i \in Const(\Delta')$, where B_i represents the bottom of the *i*-th stack and each X_i represents M_i tokens at place P_i . The *i*-th stack t_i is of the form $X_i^n \cdot B_i$, where $n \ge 0$.

For a given initial marking $\alpha_0 = (p_1, \ldots, p_k)$ of a PN Δ we construct the initial state $\beta_0 = (1, m_1, \ldots, m_k) t_1 \| \cdots \| t_k$ of the sePA Δ' , where denoting $n_i = max(0, (p_i \text{ div } M_i) - 1)$ we put $m_i = p_i - n_i \cdot M_i$ and $t_i = X_i^{n_i} \cdot B_i$. In other words we have $p_i = m_i + n_i \cdot M_i$ and moreover $m_i \in \{M_i, \ldots, 2M_i - 1\}$ if p_i is big enough (i.e. $p_i \ge M_i$).

For each transition $(l_1, \ldots, l_k) \xrightarrow{a} (r_1, \ldots, r_k)$ of the PN Δ we construct the set of sePA rules $(s, m_1, \ldots, m_k) t \xrightarrow{a} (s', m'_1, \ldots, m'_k) t'$ such that they obey the following conditions:

- Update controller conditions: $s, s' \in \{1, \dots, k\}$ and $s' = (s \mod k) + 1$.
- The general conditions for pockets $(1 \le i \le k)$:
 - $m_i, m'_i \in \{0, \ldots, 3M_i 1\},\$
 - $m_i \ge l_i$ (i.e. the transition can be performed),
 - if $i \neq s$ then $m'_i = m_i l_i + r_i$.

We now specify m'_s and the terms t, t'. The first two *Bottom rules* are the rules for working with the empty stack. The next three *Top rules* describe the rewriting of process constant X_s . Depending on the value of $\overline{m_s} = m_s - l_s + r_s$, there are *dec*, *inc*, and *basic* variants manipulating the *s*-th stack.

Rule	t	$\overline{m_s} \in$	m'_s	t'
Bottom-basic rule	B_s	$\{0,\ldots,2M_s-1\}$	$\overline{m_s}$	B_s
Bottom-inc rule	B_s	$\{2M_s, \ldots, 3M_s - 1\}$	$\overline{m_s} - M_s$	$X_s.B_s$
Top-dec rule	X_s	$\{0, \dots, M_s - 1\}$	$\overline{m_s} + M_s$	ε
Top-basic rule	X_s	$\{M_s, \ldots, 2M_s - 1\}$	$\overline{m_s}$	X_s
Top-inc rule	X_s	$\{2M_s,\ldots,3M_s-1\}$	$\overline{m_s} - M_s$	$X_s.X_s$

Theorem 1. $PN \subsetneq sePA$ with respect to bisimulation equivalence.

Proof. (*Sketch*) Let Δ is a PN and Δ' is the sePA constructed as described above. In the following β refers to a state $(s, m_1, \ldots, m_k) X_1^{n_1} . B_1 \| \ldots \| X_k^{n_k} . B_k$ of the sePA Δ' while α refers to a marking (p_1, \ldots, p_k) of the PN Δ . We show

 $R = \{(\alpha, \beta) \mid \beta \text{ is reachable and } p_i = m_i + n_i \cdot M_i \text{ for all } i = 1, \dots k\}$

is a bisimulation relation.

Let us assume that $(\alpha, \beta) \in R$ and a transition $(l_1, \ldots, l_k) \stackrel{a}{\longrightarrow} (r_1, \ldots, r_k)$ fired in α leads to α' . Exactly one sePA rule derived from this PN transition (see the table of rules) is applicable on β . (This statement is due to the straightforward observations: if $n_i \neq 0$ then $m_i \geq L_i$, hence $p_i \geq l_i$ iff $m_i \geq l_i$.) The application of this rule on β leads to $\beta' = (s', m'_1, \ldots, m'_k) X_1^{n'_1} \cdot B_1 \parallel \ldots \parallel X_k^{n'_k} \cdot B_k$ which satisfies $m'_i + n'_i \cdot M_i = m_i + n_i \cdot M_i - l_i + r_i$ and $0 \leq m'_i < 3M_i$ for all *i*. Hence, $(\alpha', \beta') \in R$. The symmetric case proceeds in a similar way.

Note that the pair of the initial marking α_0 of the PN Δ and the initial state β_0 of the sePA Δ' is in *R*. Hence, Δ and Δ' are bisimilar. We have demonstrated that PN \subseteq sePA (with respect to bisimulation equivalence).

The strictness of this relation follows from two of the results mentioned in the introduction, namely the full Turing-power of sePA [BEH95] and the decidability of reachability for PN [May81].

We note that the sePA system constructed by our algorithm does not need to be isomorphic to the original PN system (e.g. due to the different values of the update controller).

5 Reachability for wPRS Is Decidable

In this section we show that for a given wPRS Δ and its states rt_1, st_2 it is decidable whether st_2 is reachable from rt_1 or not (recall that st_2 is reachable from rt_1 if a sequence of actions σ such that $rt_1 \xrightarrow{\sigma} st_2$ exists).

Our proof exhibits a similar structure to the proof of decidability of the reachability problem for PRS [May00]; first we reduce the general problem to the reachability problem for wPRS with rules containing at most one occurrence of a sequential or parallel operator, and then we solve this subproblem using the fact that the reachability problems for both PN and PDA are decidable [May81,Büc64]. The latter part of our proof is based on a new idea of *passive steps* presented later.

To get just a sketch of the following proof we suggest to read the definitions and statements (skipping their technical proofs). Some of them are preceded by comments that provide some intuition.

As the labels on rewrite rules are not relevant here, we omit them in this section. To distinguish between rules and rewriting sequences we use $rt_1 \succ^{\Delta} st_2$ to denote that the state st_2 is reachable from rt_1 in wPRS Δ . Further, states of weak state unit are called *weak states*.

Definition 3. Let Δ be a wPRS. A rewrite rule in Δ is parallel or sequential if it has one of the following forms:

where X, Y, Z are process constants and p, q are weak states. A rule is trivial if it is both parallel and sequential (i.e. it has the form $pX \longrightarrow qY$ or $pX \longrightarrow q\varepsilon$). A wPRS Δ is in normal form if every rewrite rule in Δ is parallel or sequential. **Lemma 1.** For a wPRS Δ , terms t_1, t_2 , and weak states r, s, there are terms t'_1, t'_2 of wPRS Δ' in normal form satisfying $rt_1 \succ^{\Delta} st_2 \iff rt'_1 \succ^{\Delta'} st'_2$. Moreover, wPRS Δ' and terms t'_1, t'_2 can be effectively constructed.

Proof. In this proof we assume that the sequential composition is left-associative. It means that the term X.Y.Z is (X.Y).Z and so its subterms are X, Y, Z, and X.Y, but not Y.Z. However, the term Y || Z is a subterm of X.(Y || Z).

Let size(t) denote the number of sequential and parallel operators in term t. Given any wPRS Δ , let k_i be the number of rules $(pt \longrightarrow qt') \in \Delta$ that are neither parallel nor sequential and $size(pt \longrightarrow qt') = i$, where $size(pt \longrightarrow qt') =$ size(t) + size(t'). Thus, Δ is in normal form iff $k_i = 0$ for every i. In this case, let n = 0. Otherwise, let n be the largest i such that $k_i \neq 0$ (n exists as the set of rules is finite). We define $norm(\Delta)$ to be the pair (n, k_n) .

We now describe a procedure transforming Δ (if it is not in normal form) into a wPRS Δ' and terms t_1, t_2 into terms t'_1, t'_2 such that $norm(\Delta') < norm(\Delta)$ (with respect to the lexicographical ordering) and $rt_1 \succ^{\Delta} st_2 \iff rt'_1 \succ^{\Delta'} st'_2$.

Let us assume that wPRS Δ is not in normal form. Then there is a rule that is neither sequential nor parallel and has the maximal *size*. Take a non-atomic and proper subterm t of this rule and replace every subterm t in Δ (i.e. in rewrite rules and initial term) and in t_1 and t_2 by a fresh constant X. Then add two rules $pX \longrightarrow pt$ and $pt \longrightarrow pX$ for each weak state p. This yields a new wPRS Δ' and terms t'_1 and t'_2 where the constant X serves as an abbreviation for the term t. By the definition of *norm* we get $norm(\Delta') < norm(\Delta)$. The correctness of our transformation remains to be demonstrated, namely that

$$rt_1 \succ^{\Delta} st_2 \iff rt'_1 \succ^{\Delta'} st'_2.$$

The implication \Leftarrow is obvious. For the opposite direction we show that every rewriting step in Δ from pl_1 to ql_2 under the rule $(pl \rightarrow ql') \in \Delta$ corresponds to a sequence of several rewriting steps in Δ' leading from pl'_1 to ql'_2 , where l'_1, l'_2 are equal to l_1, l_2 with all occurrences of t replaced by X. Let us assume the rule $pl \rightarrow ql'$ modifies a subterm t of pl_1 , and/or a subterm t appears in ql_2 after the rule application (the other cases are trivial). If the rule modifies a subterm t of l_1 there are two cases. Either l includes the whole t and then the corresponding rule in Δ' (with t replaced by X) can be applied directly on pl'_1 , or, due to the left-associativity of a sequential operator, t is not a subterm of the right part of any sequential composition in l_1 and thus the application of the corresponding rule in Δ' on pl'_1 is preceded by an application of the added rule $pX \longrightarrow pt$. The situation when t appears in ql_2 after the application of the considered rule is similar. Either l' includes the whole t and then the application of the corresponding rule in Δ' results directly in ql'_2 , or t is not a subterm of the right part of any sequential composition in l_2 and thus the application of the corresponding rule in Δ' is followed by an application of the added rule $qt \longrightarrow qX$ reaching the state ql'_2 .

By repeating this procedure we finally get a wPRS Δ'' in normal form and terms t_1'', t_2'' satisfying $rt_1 \succ^{\Delta} st_2 \iff rt_1'' \succ^{\Delta''} st_2''$.

Mayr's proof for PRS now transforms the PRS Δ in normal form into the PRS Δ' in so-called *transitive normal form* satisfying $(X \longrightarrow Y) \in \Delta'$ whenever $X \succ^{\Delta} Y$. This step employs the local effect of rewriting under sequential rules in a parallel environment and vice versa. Intuitively, whenever there is a rewriting sequence

$$X \| Y \longrightarrow (X_1.X_2) \| Y \longrightarrow (X_1.X_2) \| Z \longrightarrow X_2 \| Z$$

in a PRS in normal form, then the rewriting of each parallel component is independent in the sense that there are also rewriting sequences $X \longrightarrow X_1.X_2 \longrightarrow X_2$ and $Y \longrightarrow Z$. This does not hold for wPRS in normal form as the rewriting in one parallel component can influence the rewriting in other parallel components via a weak state unit. To get this independence back we introduce the concept of *passive steps* emulating changes of a weak state produced by the environment.

Definition 4. A finite sequence of weak state pairs $PS = \{(p_i, q_i)\}_{i=1}^n$ satisfying $p_1 > q_1 \ge p_2 > q_2 \ge \cdots \ge p_n > q_n$ is called passive steps.

Let Δ be a wPRS and PS be passive steps. By $\Delta + PS$ we denote a system Δ with an added rule $pX \longrightarrow qX$ for each (p,q) in PS and $X \in Const(\Delta)$. For all terms t_1, t_2 and weak states r, s we write

 $\begin{array}{ll} rt_1 \succ_{triv}^{\Delta+PS} st_2 & iff \quad rt_1 \succ^{\Delta+PS} st_2 \quad via \ trivial \ rules, \\ rt_1 \succ_{seq}^{\Delta+PS} st_2 & iff \quad rt_1 \succ^{\Delta+PS} st_2 \quad via \ sequential \ rules, \\ rt_1 \succ_{par}^{\Delta+PS} st_2 & iff \quad rt_1 \succ^{\Delta+PS} st_2 \quad via \ parallel \ rules. \end{array}$

Informally, $rt_1 \succ^{\Delta+PS} st_2$ means that the state rt_1 can be rewritten into state st_2 provided a weak state can be passively changed from p to q for every passive step (p,q) in *PS*. Thanks to the finiteness and 'weakness' of a weak state unit, the number of different passive steps is finite.

Definition 5. Let wPRS Δ be in normal form. If for every $X, Y \in Const(\Delta)$, weak states r, s, and passive steps PS it holds that $rX \succ^{\Delta+PS} sY \implies rX \succ^{\Delta+PS}_{triv} sY$ then Δ is in flatted normal form, $rX \succ^{\Delta+PS}_{seq} sY \implies rX \succ^{\Delta+PS}_{triv} sY$ then Δ is in sequential flatted normal form, $rX \succ^{\Delta+PS}_{seq} sY \implies rX \succ^{\Delta+PS}_{triv} sY$ then Δ is in parallel flatted normal form.

The following lemma says that it is sufficient to check reachability via sequential rules and via parallel rules in order to construct a wPRS in flatted normal form. This allows us to reduce the reachability problem for wPRS to the reachability problems for wPN and wPDA (i.e. to the reachability problems for PN and PDA).

Lemma 2. If a wPRS is in both sequential and parallel flatted normal form then it is in flatted normal form as well.

Proof. We assume the contrary and derive a contradiction. Let Δ be a wPRS in sequential and parallel flatted normal form. Let us choose passive steps *PS* and a rewriting sequence in $\Delta + PS$ leading from *rX* to *sY* such that $rX \neq_{triv}^{\Delta + PS} sY$,

the number of applications of non-trivial rewrite rules applied in the sequence is minimal, and all steps of *PS* are used during the sequence. As the wPRS Δ is in both sequential and parallel flatted normal form, $rX \neq_{seq}^{\Delta+PS} sY$ and $rX \neq_{par}^{\Delta+PS} sY$. Hence, both sequential and parallel operators occur in the rewriting sequence. There are two cases.

- 1. Assume that a sequential operator appears first. The parallel operator is then introduced by the rule in the form $pU \longrightarrow qT || S$ applied to a state pU.t, where t is a sequential term. From $q(T || S).t \succ^{\Delta+PS} sY$ and the fact that at most one process constant can be removed in one rewriting step, it follows that in the rest of the sequence considered, the term T || S is rewritten onto a process constant (say V) and a weak state (say o) first. Let PS' be passive steps of PS between weak states p and o.
- 2. Assume that a parallel operator appears first. The sequential operator is then introduced by the rule in the form $pU \longrightarrow qT.S$ applied on a state pU || t, where t is a parallel term. The rest of the sequence subsumes steps rewriting the term T.S onto a process constant (say V) and a weak state (say o). Contrary to the previous case, these steps can be interleaved with steps rewriting the parallel component t and possibly changing weak state. Let PS' be passive steps of PS (between weak states p and o) merged with the changes of weak states caused by rewriting of t.

Consequently, we have a rewriting sequence in $\Delta + PS'$ from pU to oV with fewer applications of non-trivial rewrite rules. As the number of applications of non-trivial rewrite rules used in the original sequence is minimal we get $pU \not\models_{triv}^{\Delta + PS'} oV$. This contradicts our choice of rX, sY, and PS.

Example 1. Here, we illustrate a possible change of passive steps (*PS* to *PS'*) described in the second case of the proof above. Let us consider a wPRS Δ with weak states r > p > q > t > v > o > s and the following rewrite rules

$$rX \longrightarrow pU \| Z \quad pU \longrightarrow qT.S \quad qZ \longrightarrow tY \quad vT.S \longrightarrow oV \quad oV \| Y \longrightarrow sY$$

as well as the following sequence $rX \succ^{\Delta + \{(t,v)\}} sY$, i.e.

 $r\underline{X} \longrightarrow p\underline{U} \| Z \longrightarrow q(T.S) \| \underline{Z} \longrightarrow \underline{t}(T.S) \| Y \xrightarrow{passive} v(\underline{T.S}) \| Y \longrightarrow o\underline{V} \| Y \longrightarrow sY,$ where redices are underlined. The sequence constructed due to the case 2 is as: $pU \succ^{\Delta + \{(q,t),(t,v)\}} oV$, i.e. $p\underline{U} \longrightarrow q(T.S) \xrightarrow{passive} \underline{t}(T.S) \xrightarrow{passive} v(\underline{T.S}) \longrightarrow oV.$

The following lemma employs the algorithms deciding the reachability problem for PDA and PN. Recall that the classes PDA and PN coincide with the classes of wPDA and wPN, respectively.

Lemma 3. For every wPRS Δ in normal form, terms t_1, t_2 over $Const(\Delta)$, and weak states r, s of Δ a wPRS Δ' can be constructed such that Δ' is in flatted normal form and satisfies $rt_1 \succ^{\Delta} st_2 \iff rt_1 \succ^{\Delta'} st_2$.

Proof. To obtain Δ' we enrich Δ by trivial rewrite rules transforming the system into sequential and parallel flatted normal forms, which suffices thanks to Lemma 2. Using algorithms deciding reachability for PDA and PN, our algorithm checks if there are some weak states r, s, constants $X, Y \in Const(\Delta)$, and passive steps $PS = \{(p_i, q_i)\}_{i=1}^n$ (satisfying $r \geq p_1$ and $q_n \geq s$ as weak states pairs beyond this range are of no use here) such that $rX \succ_{seq}^{\Delta + PS} sY \lor rX \succ_{par}^{\Delta + PS} sY$ and $rX \neq_{triv}^{\Delta + PS} sY$. We finish if the answer is negative. Otherwise we add to Δ rules $rX \longrightarrow p_1Z_1, q_iZ_i \longrightarrow p_{i+1}Z_{i+1}$ for $i = 1, \ldots, n-1$, and $q_nZ_n \longrightarrow sY$, where Z_1, \ldots, Z_n are fresh process constants (if n = 0 then we add just the rule $rX \longrightarrow sY$). The algorithm then repeats this procedure on the system with the added rules with one difference; the X, Y range over the constants of the original system Δ . This is sufficient as new constants occur only in trivial rules⁴. The algorithm terminates as the number of iterations is bounded by the number of pairs of states rX, sY of Δ , times the number of passive steps *PS*. The correctness follows from the fact that the added rules have no influence on reachability.

Theorem 2. The reachability problem for wPRS is decidable.

Proof. (*Sketch*) Let Δ be a wPRS with states rt_1, st_2 . We want to decide whether $rt_1 \succ^{\Delta} st_2$ or not. Clearly $rt_1 \succ^{\Delta} st_2 \iff rX \succ^{\Delta'} sY$, where X, Y are fresh constants and Δ' arises from Δ by the addition of the rules $rX \longrightarrow rt_1$ and $st_2 \longrightarrow sY^5$. Hence we can directly assume that t_1, t_2 are process constants, say X, Y. Lemma 1 and Lemma 3 successively reduce the question whether $rX \succ^{\Delta} sY$ to the question whether $rX \succ^{\Delta'} sY$, where Δ' is in flatted normal form – note that Lemma 1 does not change terms t_1, t_2 if they are process constants. The definition of flatted normal form implies $rX \succ^{\Delta'} sY \iff rX \succ^{\Delta'}_{triv} sY$. Finally the relation $rX \succ^{\Delta'}_{triv} sY$ is easy to check.

6 Conclusions

We have unified a view on some (non-conservative) extensions of Process Rewrite Systems. Comparing (up to bisimulation equivalence) the mutual expressiveness of the respective subclasses, we have added some new strict relations, including the class of Petri nets being less expressive than the class of PA processes extended with a finite state control unit. Finally, we have shown that a weak state unit extension (and thus a finite constraint system extension as well) of process rewrite systems keep the reachability problem decidable.

Acknowledgements. We would like to thank Hans Hüttel and Jiří Srba for discussions, comments, and pointers; and Luca Aceto for invaluable suggestions.

⁴ If the system with added rules is not in sequential or parallel flatted normal form, then there is a counterexample with the constants *X*, *Y* of the original system Δ .

⁵ If $t_2 = \varepsilon$ then this is not a correct rule. In this case we need to add to Δ' a rule $pt \longrightarrow qY$ for each rule $(pt \longrightarrow q\varepsilon) \in \Delta$.

References

- [BCMS01] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. In *Handbook of Process Algebra*, pages 545–623. Elsevier, 2001.
- [BEH95] A. Bouajjani, R. Echahed, and P. Habermehl. On the verification problem of nonregular properties for nonregular processes. In *LICS*'95. IEEE, 1995.
- [BET03] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. *International Journal on Foundations of Computer Science*, 14(4):551–582, 2003.
- [BT03] A. Bouajjani and T. Touili. Reachability Analysis of Process Rewrite Systems. In *Proc. of FST&TCS-2003*, volume 2914 of *LNCS*, pages 74– 87. Springer, 2003.
- [Büc64] J. R. Büchi. Regular canonical systems. Arch. Math. Logik u. Grundlagenforschung, 6:91–111, 1964.
- [Cau92] D. Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106:61–86, 1992.
- [DAC98] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proc. 2nd Work-shop on Formal Methods in Software Practice (FMSP-98)*, pages 7–15, New York, 1998. ACM Press.
- [DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2): 198–208, 1983.
- [Esp02] J. Esparza. Grammars as processes. In *Formal and Natural Computing*, volume 2300 of *LNCS*. Springer, 2002.
- [HS04] H. Hüttel and J. Srba. Recursion vs. replication in simple cryptographic protocols. Submitted for publication, 2004.
- [Jan95] P. Jančar. High Undecidability of Weak Bisimilarity for Petri Nets. In Proc. of TAPSOFT, volume 915 of LNCS, pages 349–363. Springer, 1995.
- [JKM01] P. Jančar, A. Kučera, and R. Mayr. Deciding bisimulation-like equivalences with finite-state processes. *Theoretical Computer Science*, 258:409–433, 2001.
- [KŘS03] M. Křetínský, V. Řehák, and J. Strejček. Process Rewrite Systems with Weak Finite-State Unit. Technical Report FIMU-RS-2003-05, Masaryk University Brno, 2003. to appear in ENTCS as Proc. of INFINITY 03.
- [LS98] D. Lugiez and Ph. Schnoebelen. The regular viewpoint on PA-processes. In Proc. of CONCUR'98, volume 1466 of LNCS, pages 50–66, 1998.
- [May81] E. W. Mayr. An algorithm for the general Petri net reachability problem. In Proc. of 13th Symp. on Theory of Computing, pages 238–246. ACM, 1981.
- [May00] R. Mayr. Process rewrite systems. *Information and Computation*, 156(1):264–286, 2000.
- [Mil89] R. Milner. Communication and Concurrency. Prentice-Hall, 1989.
- [Mol96] F. Moller. Infinite results. In Proc. of CONCUR'96, volume 1119 of LNCS, pages 195–216. Springer, 1996.
- [Mol98] F. Moller. Pushdown Automata, Multiset Automata and Petri Nets, MFCS Workshop on concurrency. *Electronic Notes in Theoretical Computer Science*, 18, 1998.
- [MSS92] D. Muller, A. Saoudi, and P. Schupp. Alternating automata, the weak monadic theory of trees and its complexity. *Theoret. Computer Science*, 97(1–2):233–244, 1992.

- [Pet81] J. L. Peterson. *Petri Net Theory and the Modelling of Systems*. Prentice-Hall, 1981.
- [SR90] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In Proc. of 17th POPL, pages 232–245. ACM Press, 1990.
- [Srb02] J. Srba. Roadmap of infinite results. *EATCS Bulletin*, (78):163–175, 2002. http://www.brics.dk/~srba/roadmap/.
- [Str02] J. Strejček. Rewrite systems with constraints, EXPRESS'01. *Electronic Notes in Theoretical Computer Science*, 52, 2002.

A General Approach to Comparing Infinite-State Systems with Their Finite-State Specifications

Antonín Kučera^{1*} and Philippe Schnoebelen²

 ¹ Faculty of Informatics, Masaryk University, Botanická 68a, 60200 Brno, Czech Republic tony@fi.muni.cz
 ² LSV, ENS de Cachan & CNRS UMR 8643, 61, av. Pdt. Wilson, 94235 Cachan Cedex, France phs@lsv.ens-cachan.fr

Abstract. We introduce a generic family of behavioral relations for which the problem of comparing an arbitrary transition system to some finite-state specification can be reduced to a model checking problem against simple modal formulae. As an application, we derive decidability of several regular equivalence problems for well-known families of infinite-state systems.

1 Introduction

Verification of infinite-state models of systems is a very active field of research, see [9, 8, 5, 19, 31] for surveys of some subfields. In this area, researchers consider a large variety of models suited to different kinds of applications, and three main kinds of verification problems: (1) specific properties like reachability or termination, (2) model checking of temporal formulae, and (3) semantic equivalences or preorders between two systems. With most models, termination and reachability are investigated first. Positive results lead to investigations of more general temporal model checking problems. Regarding equivalence problems, positive decidability results exist mainly for strong bisimilarity (some milestones in the study include [3, 13, 12, 14, 11, 30]). For other behavioral equivalences, results are usually negative.

Regular Equivalence Problem. Recently, the problem of comparing some infinitestate process g with a *finite-state* specification f has been identified as an important subcase¹ of the general equivalence checking problem [19]. Indeed,

^{*} On leave at LSV, ENS de Cachan, France. Supported by the Grant Agency of the Czech Republic, grant No. 201/03/1161.

¹ We refer to this subcase as "the regular equivalence problem" in the rest of this paper. For example, if we say that "regular weak bisimilarity is decidable for PA processes", we mean that weak bisimilarity is decidable between PA processes and finite-state ones.

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 371-386, 2004.

in equivalence-based verification, one usually compares a "real-life" system with an abstract behavioral specification. Faithful models of real-life systems often require features like counters, subprocess creation, or unbounded buffers, that make the model infinite-state. On the other hand, the behavioral specification is usually abstract, hence naturally finite-state. Moreover, infinite-state systems are often abstracted to finite-state systems even before applying further analytical methods. This approach naturally subsumes the question if the constructed abstraction is correct (i.e., equivalent to the original system). It quickly appeared that regular equivalence problems are computationally easier than comparing two infinite-state processes, and a wealth of positive results exist [19].

The literature offers two generic techniques for deciding regular equivalences. First, Abdulla *et al.* show how to check *regular simulation* on *well-structured* processes [2]. Their algorithm is generic because a large collection of infinite-state models are well-structured [10].

The second approach is even more general: one expresses equivalence with f via a formula φ_f of some modal logic \mathcal{L} . φ_f is called a *characteristic formula* for f wrt. the given equivalence. This reduces regular equivalence problems to more familiar model checking problems. It entails decidability of regular equivalences for all systems where model checking with the logic \mathcal{L} is decidable. It is easy to give characteristic formulae wrt. bisimulation-like equivalences if one uses the modal μ -calculus [32, 26]. Browne *et al.* constructed characteristic formulae wrt. bisimilarity and branching-bisimilarity in the logic CTL [7]. Unfortunately, CTL (or μ -calculus) model checking is undecidable on many process classes like PA, Petri nets, lossy channel systems, etc. Later, it has been shown that characteristic formulae wrt. strong and weak bisimilarity can be constructed even in the $\mathcal{L}(\mathbf{EX}_{\alpha}, \mathbf{EF}, \mathbf{EF}_{\tau})$ fragment of CTL [15]. This logic is sufficiently simple and its associated model-checking problem is decidable in many classes of infinite-state systems (including PA, lossy channel systems, and pushdown automata) [24].

Our Contribution. We introduce *full regular equivalences,* a variant of regular equivalences, and develop a generic approach to the reduction of full regular equivalences to model checking (essentially) the EF fragment of modal logic². Compared to regular equivalences, full regular equivalence has the additional requirement that the state-space of the infinite system must be included in the state-space of the finite system up to the given equivalence. We argue that full regular equivalence is as natural as regular equivalence in most practical situations (additionally the two variants turn out to coincide in many cases). Moreover, an important outcome of our results is that full regular equivalence is "more decidable" than regular equivalence for trace-like and simulation-like equivalences. For example, regular trace equivalence is undecidable for BPA (and hence also for pushdown and PA processes), while full regular trace equivalence

² In fact we provide reductions to $\mathcal{L}(\mathbf{EX}_{\alpha}, \mathbf{EF}, \mathbf{EF}_{\tau})$ and to $\mathcal{L}(\mathbf{EU}_{\alpha}, \mathbf{EF})$, two different fragments of modal logic that have incomparable expressive power.

is decidable for these models. Similar examples can be given for simulation-like equivalences. See Section 2 and Section 6 for further comments.

We offer two main reductions. One applies to a large parameterized family of equivalences defined via a transfer property (we call them MTB equivalences). The other applies to a large parameterized family of equivalences based on sets of enriched traces (we call them PQ equivalences). Together they cover virtually all process equivalences used in verification [33]. For all of these, full regular equivalence with some f is reduced to EF model-checking, hence shown decidable for a large family of infinite-state models. More precisely, the constructions output a *characteristic formula* for f wrt. a given equivalence, which expresses the property of "being fully equivalent to f". In particular, this works for bisimulation-like equivalences (weak, delay, early, branching), and thus we also obtain a refinement of the result presented in [7] which says that a characteristic formula wrt. branching bisimilarity is constructible in CTL. The main "message" of this part is that full regular equivalence is decidable for many more semantic equivalences and classes of infinite-state models than regular equivalence. In this paper we do not aim to develop specific methods for particular models and equivalences. (Such methods can be more efficient than our generic (modelindependent) algorithm-for example, it has recently been shown in [20] that full regular equivalence with PDA processes can be decided by a PDA-specific algorithm which needs only polynomial time for some MTB equivalences and some subclasses of PDA processes.)

Another contribution of this paper is a model-checking algorithm for the logic $\mathcal{L}(\mathbf{EX}_{\alpha}, \mathbf{EF}, \mathbf{EF}_{\tau}, \mathbf{EU}_{\alpha})$ and lossy channel systems. This allows one to apply the previous abstract results also to processes of lossy channel systems (for other models like, e.g., pushdown automata, PA processes, or PAD processes, the decidability of model-checking problem with the logic EF is already known).

Due to space constraints, we had to omit all proofs. These can be found in a full version of this paper [21].

2 (Full) Regular Equivalence

We start by recalling basic definitions. Let $Act = \{a, b, c, ...\}$ be a countably infinite set of *actions*, and let $\tau \notin Act$ be a distinguished *silent action*. For $A \subseteq Act$, A_{τ} denotes the set $A \cup \{\tau\}$. We use $\alpha, \beta, ...$ to range over Act_{τ} . A *transition system* is a triple $T = (S, \rightarrow, A)$ where S is a set of *states*, $A \subset Act_{\tau}$ is a finite *alphabet*, and $\rightarrow \subseteq S \times A \times S$ is a *transition relation*. We write $s \stackrel{\alpha}{\rightarrow} t$ instead of $(s, \alpha, t) \in \rightarrow$, and we extend this notation to elements of A^* in the standard way. We say that a state t is *reachable* from a state s, written $s \rightarrow^* t$, if there is $w \in A^*$ such that $s \stackrel{w}{\rightarrow} t$. Further, for every $\alpha \in Act_{\tau}$ we define the relation $\stackrel{\alpha}{\Rightarrow} \subseteq S \times S$ as follows: $s \stackrel{\alpha}{\Rightarrow} t$ iff there is a sequence of the form $s = p_0 \stackrel{\tau}{\rightarrow} \cdots \stackrel{\tau}{\rightarrow} p_k = t$ where $k \ge 0$; $s \stackrel{\alpha}{\Rightarrow} t$ where $a \neq \tau$ iff there are p, q such that $s \stackrel{\pi}{\Rightarrow} p \stackrel{\alpha}{\Rightarrow} q \stackrel{\pi}{\Rightarrow} t$. From now on, a *process* is formally understood as a state of (some) transition system. Intuitively, transitions from a given process s model possible computational steps, and the silent action τ is used to mark those steps which are internal (i.e., not externally observable). Since we sometimes consider processes without explicitly defining their associated transition systems, we also use $\mathcal{A}(s)$ to denote the alphabet of (the underlying transition system of) the process s. A process s is τ -free if $\tau \notin \mathcal{A}(s)$.

Let ~ be an arbitrary process equivalence, g a (general) process, \mathcal{F} a finitestate system, and f a process of \mathcal{F} .

Definition 1 (Full Regular Equivalence). We say g is fully equivalent to f (in \mathcal{F}) iff:

- $-g \sim f$ (g is equivalent to f), and
- for all $g \to g'$, there is some f' in \mathcal{F} s.t. $g' \sim f'$ (every process reachable from g has an equivalent in \mathcal{F}).

Observe that the equivalent f' does *not* have to be reachable from f.

In verification settings, requiring that some process g is fully equivalent to a finite-state specification \mathcal{F} puts some additional constraints on g: its whole state-space must be accounted for in a finite way. To get some intuition why this is meaningful, consider, e.g., the finite-state system with three states f, f', f''and transitions $f \xrightarrow{a} f, f' \xrightarrow{a} f''$. Suppose that all transitions of a given infinitestate system g are labeled by a. Then regular trace equivalence to f means that g can do infinitely many a's (assuming that g is finitely branching), while full regular trace equivalence to f means that g can do infinitely many a's and whenever it decides to terminate, it can reach a terminated state in at most one transition. This property cannot be encoded as regular bisimulation equivalence or regular simulation equivalence by any finite-state system. Let us also note that when \sim is an equivalence of the bisimulation family, then regular equivalence is automatically "full".

3 MTB Preorder and Equivalence

In this paper, we aim to prove general results about equivalence-checking between infinite-state and finite-state processes. To achieve that, we consider an abstract notion of process preorder and process equivalence which will be introduced next.

A transfer is one of the three operators on binary relations defined as follows: sim(R) = R, $bisim(R) = R \cap R^{-1}$, $contrasim(R) = R^{-1}$. A mode is a subset of $\{\eta, d\}$ (the η and d are just two different symbols). A basis is an equivalence over processes satisfying the following property: whenever $(s, u) \in B$ and $s \stackrel{\tau}{\Rightarrow} t \stackrel{\tau}{\Rightarrow} u$, then also $(s, t) \in B$.

Definition 2. Let S be a binary relation over processes and M a mode. A move $s \stackrel{\alpha}{\Rightarrow} t$ is tightly S-consistent with M if either $\alpha = \tau$ and s = t, or there is a sequence $s = s_0 \stackrel{\tau}{\to} \cdots \stackrel{\tau}{\to} s_k \stackrel{\alpha}{\to} t_0 \stackrel{\tau}{\to} \cdots \stackrel{\tau}{\to} t_{\ell} = t$, where $k, \ell \ge 0$, such that the

following holds: (1) if $\eta \in M$, then $(s_i, s_j) \in S$ for all $0 \le i, j \le k$; (2) if $d \in M$, then $(t_i, t_j) \in S$ for all $0 \le i, j \le \ell$.

The loose S-consistency of $s \stackrel{\alpha}{\Rightarrow} t$ with M is defined in the same way, but the conditions (1), (2) are weakened—we only require that $(s_0, s_k), (s_k, s_0) \in S$, and $(t_0, t_\ell), (t_\ell, t_0) \in S$.

Definition 3. Let T be a transfer, M a mode, and B a basis. A binary relation \mathcal{R} over processes is a tight (or loose) MTB-relation if it satisfies the following:

- $-\mathcal{R}\subseteq B$
- whenever $(p,q) \in \mathcal{R}$, then for every tightly (or loosely, resp.) \mathcal{R} -consistent move $p \stackrel{\alpha}{\Rightarrow} p'$ there is a tightly (or loosely, resp.) \mathcal{R} -consistent move $q \stackrel{\alpha}{\Rightarrow} q'$ such that $(p',q') \in T(\mathcal{R})$.

We write $s \sqsubseteq t$ (or $s \preccurlyeq t$, resp.), if there is a tight (or loose, resp.) MTBrelation \mathcal{R} such that $(s,t) \in \mathcal{R}$. We say that s,t are tightly (or loosely, resp.) MTB-equivalent, written $s \sim t$ (or $s \approx t$, resp.), if $s \sqsubseteq t$ and $t \sqsubseteq s$ (or $s \preccurlyeq t$ and $t \preccurlyeq s$, resp.).

It is standard that such a definition entails that \sqsubseteq and \preccurlyeq are preorders, and ~ and \approx are equivalences over the class of all processes. The relationship between \sqsubseteq and \preccurlyeq relations is clarified in the next lemma (this is where we need the defining property of a base).

Lemma 1. We have that $\sqsubseteq = \measuredangle$ (and hence also $\sim = \approx$).

Before presenting further technical results, let us briefly discuss and justify the notion of MTB equivalence. The class of all MTB equivalences can be partitioned into the subclasses of simulation-like, bisimulation-like, and contrasimulation-like equivalences according to the chosen transfer. Additional conditions which must be satisfied by equivalent processes can be specified by an appropriately defined base. For example, we can put *B* to be *true*, *ready*, or *terminate* where

- $-(s,t) \in true$ for all s and t;
- $-(s,t) \in ready \text{ iff } \{a \in Act_{\tau} \mid \exists s' : s \stackrel{a}{\Rightarrow} s'\} = \{a \in Act_{\tau} \mid \exists t' : t \stackrel{a}{\Rightarrow} t'\};$
- $-(s,t) \in terminate \text{ iff } s \text{ and } t \text{ are either both terminating, or both non-terminating (a process p is terminating iff <math>p \stackrel{\alpha}{\Rightarrow} p'$ implies $\alpha = \tau$ and p cannot perform an infinite sequence of τ -transitions).

The mode specifies the level of 'control' over the states that are passed through by $\stackrel{\alpha}{\Rightarrow}$ transitions. In particular, by putting T = bisim, B = true, and choosing M to be \emptyset , { η }, {d}, or { η , d}, one obtains weak bisimilarity [25], η -bisimilarity [4], delay-bisimilarity, and branching bisimilarity [34], respectively³.

³ Our definition of *MTB* equivalence does not directly match the definitions of η -, delay-, and branching bisimilarity that one finds in the literature. However, it is easy to show that one indeed yields exactly these equivalences.

"Reasonable" refinements of these bisimulation equivalences can be obtained by redefining *B* to something like *terminate*—sometimes there is a need to distinguish between, e.g., terminated processes and processes which enter an infinite internal loop. If we put T = sim, B = true, and $M = \emptyset$, we obtain weak simulation equivalence; and by redefining *B* to *ready* we yield a variant of ready simulation equivalence. The equivalence where T = contrasim, B = true, and $M = \emptyset$ is known as contrasimulation (see, e.g., [35])⁴.

The definition of MTB equivalence allows to combine all of the three parameters arbitrarily, and our results are valid for all such combinations (later we adopt some natural effectiveness assumptions about B, but this will be the only restriction).

Definition 4. For every $k \in \mathbb{N}_0$, the binary relations \sqsubseteq_k , \sim_k , \preccurlyeq_k , and \approx_k are defined as follows: $s \sqsubseteq_0 t$ iff $(s,t) \in B$; $s \sqsubseteq_{k+1} t$ iff $(s,t) \in B$ and for every tightly \sqsubseteq_k -consistent move $s \stackrel{\simeq}{\Rightarrow} s'$ there is some tightly \sqsubseteq_k -consistent move $t \stackrel{\simeq}{\Rightarrow} t'$ such that $(s', t') \in T(\sqsubseteq_k)$.

The \preccurlyeq_k relations are defined in the same way, but we require only loose \preccurlyeq_k consistency of moves in the inductive step. Finally, we put $s \sim_k t$ iff $s \sqsubseteq_k t$ and $t \sqsubseteq_k s$, and similarly $s \approx_k t$ iff $s \preccurlyeq_k t$ and $t \preccurlyeq_k s$.

A trivial observation is that $\preccurlyeq_k \supseteq \preccurlyeq_{k+1} \supseteq \preccurlyeq$, $\sqsubseteq_k \supseteq \sqsubseteq_{k+1} \supseteq \sqsubseteq$, $\sim_k \supseteq \sim_{k+1} \supseteq \sim$, and $\approx_k \supseteq \approx_{k+1} \supseteq \approx$ for each $k \in \mathbb{N}_0$. In general, $\sqsubseteq_k \neq \preccurlyeq_k$; however, if we restrict ourselves to processes of some fixed finite-state system, we can prove the following:

Lemma 2. Let $\mathcal{F} = (F, \rightarrow, \mathcal{A})$ be a finite-state system with n states. Then $\sqsubseteq_{n^2-1} = \bigsqcup_{n^2} = \bigsqcup_{n^2} = \preccurlyeq \preccurlyeq = \preccurlyeq_{n^2-1} = \preccurlyeq_{n^2}$, where all of the relations are considered as being restricted to $F \times F$.

Theorem 1. Let $\mathcal{F} = (F, \rightarrow, \mathcal{A})$ be a finite-state system with n states, f a process of F, and g some (arbitrary) process. Then the following three conditions are equivalent.

(a) $g \sim f$ and for every $g \to^* g'$ there is some $f' \in F$ such that $g' \sim f'$. (b) $g \sim_{n^2} f$ and for every $g \to^* g'$ there is some $f' \in F$ such that $g' \sim_{n^2} f'$. (c) $g \approx_{n^2} f$ and for every $g \to^* g'$ there is some $f' \in F$ such that $g' \approx_{n^2} f'$.

3.1 Encoding MTB Equivalence into Modal Logic

In this section we show that the conditions (b) and (c) of Theorem 1 can be expressed in modal logic. Let us consider a class of modal formulae defined by the following abstract syntax equation (where α ranges over Act_{τ}):

⁴ Contrasimulation can also be seen as a generalization of coupled simulation [27, 28], which was defined only for the subclass of divergence-free processes (where it coincides with contrasimulation). It is worth to note that contrasimulation coincides with strong bisimilarity on the subclass of τ -free processes (to see this, realize that one has to consider the moves $s \stackrel{\sim}{\Rightarrow} s$ even if s is τ -free). This is (intuitively) the reason why contrasimulation has some nice properties also in the presence of silent moves.

$\varphi ::= \mathsf{tt} | \varphi_1 \land \varphi_2 | \neg \varphi | \mathbf{EX}_{\alpha} \varphi | \mathbf{EF} \varphi | \mathbf{EF}_{\tau} \varphi | \varphi_1 \mathbf{EU}_{\alpha} \varphi_2$

The semantics (over processes) is defined inductively as follows:

- -s = tt for every process s.
- $\begin{array}{ll} -s \models \varphi_1 \land \varphi_2 & \text{iff } s \models \varphi_1 \text{ and } s \models \varphi_2. \\ -s \models \neg \varphi & \text{iff } s \not\models \varphi. \end{array}$
- $\begin{array}{ll} -s \models \mathbf{E} \mathbf{X}_{\alpha} \varphi & \text{iff there is } s \xrightarrow{\alpha} s' \text{ such that } s' \models \varphi. \\ -s \models \mathbf{E} \mathbf{F} \varphi & \text{iff there is } s \xrightarrow{\gamma} s' \text{ such that } s' \models \varphi. \\ -s \models \mathbf{E} \mathbf{F}_{\tau} \varphi & \text{iff there is } s \xrightarrow{\tau} s' \text{ such that } s' \models \varphi. \end{array}$

- $-s \models \varphi_1 \mathbf{EU}_{\alpha} \varphi_2$ iff either $\alpha = \tau$ and $s \models \varphi_2$, or there is a sequence $s = s_0 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_m \xrightarrow{\alpha} s'$, where $m \ge 0$, such that $s_i \models \varphi_1$ for all $0 \le i \le m$ and $s' \models \varphi_2.$

The dual operator to **EF** is **AG**, defined by $\mathbf{AG} \varphi \equiv \neg \mathbf{EF} \neg \varphi$.

Let M_1, \ldots, M_k range over $\{\mathbf{EX}_{\alpha}, \mathbf{EF}, \mathbf{EF}_{\tau}, \mathbf{EU}_{\alpha}\}$. The (syntax of the) logic $\mathcal{L}(M_1,\ldots,M_k)$ consists of all modal formulae built over the modalities M_1,\ldots,M_k .

Let ~ be an *MTB* equivalence. Our aim is to show that for every finite f there are formulae φ_f of $\mathcal{L}(\mathbf{EF}, \mathbf{EU}_{\alpha})$ and ψ_f of $\mathcal{L}(\mathbf{EX}_{\alpha}, \mathbf{EF}, \mathbf{EF}_{\tau})$ such that for every process g where $\mathcal{A}(g) \subseteq \mathcal{A}$ we have that $g \models \varphi_f$ (or $g \models \psi_f$) iff the processes g and f satisfy the condition (b) (or (c), resp.) of Theorem 1. Clearly such formulae cannot always exist without some additional assumptions about the base B. Actually, all we need is to assume that the equivalence B with processes of a given finite-state system $\mathcal{F} = (F, \rightarrow, \mathcal{A})$ is definable in the aforementioned logics. More precisely, for each $f \in F$ there should be formulae Ξ_f^t and Ξ_f^t of the logics $\mathcal{L}(\mathbf{EF}, \mathbf{EU}_{\alpha})$ and $\mathcal{L}(\mathbf{EX}_{\alpha}, \mathbf{EF}, \mathbf{EF}_{\tau})$, respectively, such that for every process g where $\mathcal{A}(g) \subseteq \mathcal{A}$ we have that $(g, f) \in B$ iff $g \models \Xi_f^t$ iff $g \models \Xi_f^\ell$. Since we are also interested in complexity issues, we further assume that the formulae Ξ_f^t and Ξ_f^{ℓ} are *efficiently* computable from \mathcal{F} . An immediate consequence of this assumption is that *B* over $F \times F$ is efficiently computable. This is because the model-checking problem with $\mathcal{L}(\mathbf{EF}, \mathbf{EU}_{\alpha})$ and $\mathcal{L}(\mathbf{EX}_{\alpha}, \mathbf{EF}, \mathbf{EF}_{\tau})$ is decidable in polynomial time over finite-state systems. To simplify the presentation of our complexity results, we adopt the following definition:

Definition 5. We say that a base B is well-defined if there is a polynomial \mathcal{P} (in two variables) such that for every finite-state system $\mathcal{F} = (F, \rightarrow, \mathcal{A})$ the set $\{\Xi_f^t, \Xi_f^\ell \mid f \in F\}$ can be computed, and the relation $B \cap (F \times F)$ can be decided, in time $\mathcal{O}(\mathcal{P}(|F|, |\mathcal{A}|))$.

Remark 1. Note that a well-defined *B* is not necessarily decidable over process classes which contain infinite-state processes-for example, the ready base introduced in the previous section is well-defined but it is not decidable for, e.g., CCS processes. In fact, the Ξ_f^t formulae are only required for the construction of φ_f , and the Ξ_f^{ℓ} formulae are required only for the construction of ψ_f . (This is

why we provide two different formulae for each f.) Note that there are bases for which we can construct only one of the Ξ_f^t and Ξ_f^ℓ families, which means that for some *MTB* equivalences we can construct only one of the φ_f and ψ_f formulae. A concrete example is the *terminate* base of the previous section, which is definable in $\mathcal{L}(\mathbf{EX}_{\alpha}, \mathbf{EF}, \mathbf{EF}_{\tau})$ but not in $\mathcal{L}(\mathbf{EF}, \mathbf{EU}_{\alpha})$.

For the rest of this section, we fix some *MTB*-equivalence ~ where *B* is well-defined, and a finite-state system $\mathcal{F} = (F, \rightarrow, \mathcal{A})$ with *n* states.

Let $\langle \alpha, \varphi_{\eta}, \varphi_{d} \rangle^{t}$ and $\langle \alpha, \varphi_{\eta}, \varphi_{d} \rangle^{\ell}$ be unary modal operators whose semantics is defined as follows:

- $-s \models \langle \alpha, \varphi_{\eta}, \varphi_{d} \rangle^{t} \varphi \text{ iff either } \alpha = \tau \text{ and } s \models \varphi, \text{ or there is a sequence of the form } s = p_{0} \xrightarrow{\tau} \cdots p_{k} \xrightarrow{\alpha} q_{0} \xrightarrow{\tau} \cdots \xrightarrow{\tau} q_{m}, \text{ where } k, m \ge 0, \text{ such that } p_{i} \models \varphi_{\eta} \text{ for all } 0 \le i \le k, q_{j} \models \varphi_{d} \text{ for all } 0 \le j \le m, \text{ and } q_{m} \models \varphi.$
- $-s \models \langle \alpha, \varphi_{\eta}, \varphi_{d} \rangle^{\ell} \varphi \text{ iff either } \alpha = \tau \text{ and } s \models \varphi, \text{ or there is a sequence of the form } s = p_{0} \xrightarrow{\tau} \cdots p_{k} \xrightarrow{\alpha} q_{0} \xrightarrow{\tau} \cdots \xrightarrow{\tau} q_{m}, \text{ where } k, m \ge 0, \text{ such that } p_{0} \models \varphi_{\eta}, p_{k} \models \varphi_{\eta}, q_{0} \models \varphi_{d}, q_{m} \models \varphi_{d}, \text{ and } q_{m} \models \varphi.$

We also define $[\alpha, \varphi_{\eta}, \varphi_d]^t \varphi$ as an abbreviation for $\neg \langle \alpha, \varphi_{\eta}, \varphi_d \rangle^t \neg \varphi$, and similarly $[\alpha, \varphi_{\eta}, \varphi_d]^{\ell} \varphi$ is used to abbreviate $\neg \langle \alpha, \varphi_{\eta}, \varphi_d \rangle^{\ell} \neg \varphi$.

Lemma 3. The $\langle \alpha, \varphi_{\eta}, \varphi_{d} \rangle^{t}$ and $\langle \alpha, \varphi_{\eta}, \varphi_{d} \rangle^{\ell}$ modalities are expressible in $\mathcal{L}(\mathbf{EU}_{\alpha})$ and $\mathcal{L}(\mathbf{EX}_{\alpha}, \mathbf{EF}_{\tau})$, respectively:

Since the conditions (b) and (c) of Theorem 1 are encoded into $\mathcal{L}(\mathbf{EF}, \mathbf{EU}_{\alpha})$ and $\mathcal{L}(\mathbf{EX}_{\alpha}, \mathbf{EF}, \mathbf{EF}_{\tau})$ along the same scheme, we present both constructions at once by adopting the following notation: $\langle \alpha, \varphi_{\eta}, \varphi_{d} \rangle$ stands either for $\langle \alpha, \varphi_{\eta}, \varphi_{d} \rangle^{t}$ or $\langle \alpha, \varphi_{\eta}, \varphi_{d} \rangle^{\ell}$, \mathcal{Z}_{f} denotes either \mathcal{Z}_{f}^{t} or \mathcal{Z}_{f}^{ℓ} , $\overset{\circ}{=}_{k}$ denotes either \sim_{k} or \approx_{k} , and \leqslant_{k} denotes either \sqsubseteq_{k} or \preccurlyeq_{k} , respectively. Moreover, we write $s \overset{\alpha,k}{\longrightarrow} t$ to denote that there is either a tightly \sqsubseteq_{k} -consistent move $s \overset{\alpha}{\Rightarrow} t$, or a loosely \preccurlyeq_{k} -consistent move $s \overset{\alpha}{\Rightarrow} t$, respectively.

Definition 6. For all $f \in F$ and $k \in \mathbb{N}_0$ we define the formulae $\Phi_{f,k}$, $\Psi_{f,k}$, and $\Theta_{f,k}$ inductively as follows:

$$\begin{aligned} &-\Phi_{f,0} = \Psi_{f,0} = \Xi_f \\ &-\Theta_{f,k} = \Phi_{f,k} \wedge \Psi_{f,k} \\ &-\Phi_{f,k+1} = \Xi_f \wedge (\mathbf{AG} \bigvee_{f' \in F} \Theta_{f',k}) \wedge (\bigwedge_{f \xrightarrow{\alpha,k}{\longrightarrow} f'} (\bigvee_{f_1, f_2 \in F} \langle \alpha, \varphi_{f_1,k}, \psi_{f_2,k} \rangle \xi_{f',k})) \\ &-\Psi_{f,k+1} = \Xi_f \wedge (\mathbf{AG} \bigvee_{f' \in F} \Theta_{f',k}) \wedge \bigwedge_{\alpha \in \mathcal{A}_{\tau}, f_1, f_2 \in F} ([\alpha, \varphi_{f_1,k}, \psi_{f_2,k}] (\bigvee_{f \xrightarrow{\alpha,k}{\longrightarrow} f'} \varrho_{f',k})) \end{aligned}$$

where

- if $\eta \in M$, then $\varphi_{f_1,k} = \Theta_{f_1,k}$, otherwise $\varphi_{f_1,k} = \text{tt}$; - if $d \in M$, then $\psi_{f_2,k} = \Theta_{f_2,k}$, otherwise $\psi_{f_2,k} = \text{tt}$; - if T = sim, then $\xi_{f',k} = \Phi_{f',k}$ and $\varrho_{f',k} = \Psi_{f',k}$; - if T = bisim, then $\xi_{f',k} = \varrho_{f',k} = \Theta_{f',k}$; - if T = contrasim, then $\xi_{f',k} = \Psi_{f',k}$ and $\varrho_{f',k} = \Phi_{f',k}$.

The empty conjunction is equivalent to tt, and the empty disjunction to ff.

The meaning of the constructed formulae is explained in the next theorem. Intuitively, what we would like to have is that for every process g where $\mathcal{A}(g) \subseteq \mathcal{A}$ it holds that $g \models \Phi_{f,k}$ iff $f \leq_k g$, and $g \models \Psi_{f,k}$ iff $g \leq_k f$. However, this is (provably) not achievable—the \leq_k preorder with a given finite-state process is not directly expressible in the logics $\mathcal{L}(\mathbf{EF}, \mathbf{EU}_{\alpha})$ and $\mathcal{L}(\mathbf{EX}_{\alpha}, \mathbf{EF}, \mathbf{EF}_{\tau})$. The main trick (and subtlety) of the presented inductive construction is that the formulae $\Phi_{f,k}$ actually express stronger conditions.

Theorem 2. Let g be an (arbitrary) process such that $\mathcal{A}(g) \subseteq \mathcal{A}$. Then for all $f \in F$ and $k \in \mathbb{N}_0$ we have the following:

- (a) $g \models \Phi_{f,0}$ iff $f \leq_0 g$; further, $g \models \Phi_{f,k+1}$ iff $f \leq_{k+1} g$ and for each $g \rightarrow^* g'$ there is $f' \in F$ such that $g' \stackrel{\circ}{=}_k f'$.
- (b) $g \models \Psi_{f,0}$ iff $g \leq_0 f$; further, $g \models \Psi_{f,k+1}$ iff $g \leq_{k+1} f$ and for each $g \rightarrow^* g'$ there is $f' \in F$ such that $g' \stackrel{\circ}{=}_k f'$.
- (c) $g \models \Theta_{f,0}$ iff $g \stackrel{\circ}{=}_0 f$; further, $g \models \Theta_{f,k+1}$ iff $f \stackrel{\circ}{=}_{k+1} g$ and for each $g \rightarrow^* g'$ there is $f' \in F$ such that $g' \stackrel{\circ}{=}_k f'$.

In general, the \leq_k -consistency of moves $g \stackrel{\alpha}{\Rightarrow} g'$ can be expressed in a given logic only if one can express the $\stackrel{\alpha}{=}_k$ equivalence with g and g'. Since g and g'can be infinite-state processes, this is generally impossible. This difficulty was overcome in Theorem 2 by using the assumption that g and g' are $\stackrel{\alpha}{=}_k$ equivalent to some f_1 and f_2 of F. Thus, we only needed to encode the $\stackrel{\alpha}{=}_k$ equivalence with f_1 and f_2 which is (in a way) achieved by the $\Theta_{f_1,k}$ and $\Theta_{f_2,k}$ formulae. An immediate consequence of Theorem 1 and Theorem 2 is the following:

Corollary 1. Let g be an (arbitrary) process such that $\mathcal{A}(g) \subseteq \mathcal{A}$, and let $f \in F$. Then the following two conditions are equivalent:

(a) $g \sim f$ and for every $g \to^* g'$ there is some $f' \in F$ such that $g' \sim f'$. (b) $g \models \Theta_{f,n^2} \wedge \operatorname{AG}(\bigvee_{f' \in F} \Theta_{f',n^2})$.

Since the formula $\Theta_{f,n^2} \wedge \mathbf{AG}(\bigvee_{f' \in F} \Theta_{f',n^2})$ is effectively constructible, the problem (a) of the previous corollary is effectively reducible to the problem (b). A natural question is what is the complexity of the reduction from (a) to (b). At first glance, it seems to be exponential because the size of Θ_{f',n^2} is exponential in the size of \mathcal{F} . However, the number of distinct subformulae in Θ_{f',n^2} is only *polynomial*. This means that if we represent the formula $\Theta_{f,n^2} \wedge \mathbf{AG}(\bigvee_{f' \in F} \Theta_{f',n^2})$ by a *circuit*⁵, then the size of this circuit is only polynomial in the size of \mathcal{F} . This is important because the complexity of many model-checking algorithms actually depends on the size of the circuit representing a given formula rather than on the size of the formula itself. The size of the circuit for $\Theta_{f,n^2} \wedge \mathbf{AG}(\bigvee_{f' \in F} \Theta_{f',n^2})$ is estimated in our next lemma.

Lemma 4. The formula $\Theta_{f,n^2} \wedge \operatorname{AG}(\bigvee_{f' \in F} \Theta_{f',n^2})$ can be represented by a circuit constructible in $\mathcal{O}(n^6 \cdot |\mathcal{A}| + \mathcal{P}(n, |\mathcal{A}|))$ time.

⁵ A circuit (or a DAG) representing a formula φ is basically the syntax tree for φ where the nodes representing the same subformula are identified.

4 PQ Preorder and Equivalence

Let M, N be sets of processes. We write $M \stackrel{\alpha}{\Rightarrow} N$ iff for every $t \in N$ there is some $s \in M$ such that $s \stackrel{\alpha}{\Rightarrow} t$. In the next definition we introduce another parametrized equivalence which is an abstract template for trace-like equivalences.

Definition 7. Let P be a preorder over the class of all processes and let $Q \in \{\forall, \exists\}$. For every $i \in \mathbb{N}_0$ we inductively define the relation \sqsubseteq_i as follows:

- $-s \sqsubseteq_0 M$ for every process s and every set of processes M such that
 - if $Q = \forall$, then $(s, t) \in P$ for every $t \in M$;
 - if $Q = \exists$, then $(s, t) \in P$ for some $t \in M$;
- $-s \sqsubseteq_{i+1} M$ if $s \sqsubseteq_i M$ and for every $s \stackrel{\alpha}{\Rightarrow} t$ there is $M \stackrel{\alpha}{\Rightarrow} N$ such that $t \sqsubseteq_i N$.

Slightly abusing notation, we write $s \sqsubseteq_i t$ instead of $s \sqsubseteq_i \{t\}$. Further, we define the PQ preorder, denoted " \sqsubseteq ", by $s \sqsubseteq M$ iff $s \sqsubseteq_i M$ for every $i \in \mathbb{N}_0$. Processes s, t are PQ equivalent, written $s \sim t$, iff $s \sqsubseteq t$ and $t \sqsubseteq s$.

For every process s, let $I(s) = \{a \in Act \mid s \stackrel{a}{\Rightarrow} t \text{ for some } t\}$ (note that $\tau \notin I(s)$). Now consider the preorders T, D, F, R, S defined as follows:

- $-(s,t) \in T$ for all s,t (true).
- $-(s,t) \in D$ iff both I(s) and I(t) are either empty or non-empty (deadlock equivalence).
- $-(s,t) \in F$ iff $I(s) \supseteq I(t)$ (failure preorder).
- $-(s,t) \in R$ iff I(s) = I(t) (ready equivalence).
- $-(s,t) \in S \text{ iff } s \text{ and } t \text{ are trace equivalent (that is, iff } \{w \in Act^* \mid \exists s \stackrel{w}{\Rightarrow} s'\} = \{w \in Act^* \mid \exists t \stackrel{w}{\Rightarrow} t'\}.$

Now one can readily check that TQ, $D\exists$, $F\exists$, $F\forall$, $R\exists$, $R\forall$, and $S\exists$ equivalence is in fact trace, completed trace, failure, failure trace, readiness, ready trace, and possible futures equivalence, respectively. Other trace-like equivalences can be defined similarly.

Lemma 5. Let $\mathcal{F} = (F, \rightarrow, \mathcal{A})$ be a finite-state system with n states. Then $\sqsubseteq_{n2^n-1} = \bigsqcup_{n2^n} = \bigsqcup_{n2^n}$, where all of the relations are considered as being restricted to $F \times 2^F$.

Lemma 6. For all $i \in \mathbb{N}_0$, processes s, t, and sets of processes M, N we have that

- (a) if $s \sqsubseteq_i t$ and $t \sqsubseteq_i M$, then also $s \sqsubseteq_i M$;
- (b) if $s \sqsubseteq_i M$ and for every $u \in M$ there is some $v \in N$ such that $u \sqsubseteq_i v$, then also $s \sqsubseteq_i N$.

Theorem 3. Let $\mathcal{F} = (F, \rightarrow, A)$ be a finite-state system with n states, f a process of F, and g some (arbitrary) process. Then the following two conditions are equivalent.

(a) $g \sim f$ and for every $g \to^* g'$ there is some $f' \in F$ such that $g' \sim f'$. (b) $g \sim_{n2^n} f$ and for every $g \to^* g'$ there is some $f' \in F$ such that $g' \sim_{n2^n} f'$.

Now we show how to encode the condition (b) of Theorem 3 into modal logic. To simplify our notation, we introduce the $\langle\!\langle \alpha \rangle\!\rangle$ operator defined as follows: $\langle\!\langle \alpha \rangle\!\rangle \varphi$ stands either for $\mathbf{EF}_{\tau} \varphi$ (if $\alpha = \tau$), or $\mathbf{EF}_{\tau} \mathbf{EX}_{\alpha} \mathbf{EF}_{\tau} \varphi$ (if $\alpha \neq \tau$). Moreover, $[\![\alpha]\!] \varphi \equiv \neg \langle\!\langle \alpha \rangle\!\rangle \neg \varphi$. Similarly as in the case of *MTB* equivalence, we need some effectiveness assumptions about the preorder *P*, which are given in our next definition.

Definition 8. We say that P is well-defined if for every finite-state system $\mathcal{F} = (F, \rightarrow, \mathcal{A})$ and every $f \in F$ the following conditions are satisfied:

- There are effectively definable formulae Ξ_f , Γ_f of the logic $\mathcal{L}(\langle\!\langle \alpha \rangle\!\rangle, \mathbf{EF})$ such that for every process g where $\mathcal{A}(g) \subseteq \mathcal{A}$ we have that $g \models \Xi_f$ iff $(f,g) \in P$, and $g \models \Gamma_f$ iff $(g, f) \in P$.
- There is a polynomial \mathcal{P} (in two variables) such that for every finite-state system $\mathcal{F} = (F, \rightarrow, \mathcal{A})$ the set $\{\Xi_f, \Gamma_f \mid f \in F\}$ can be computed, and the relation $P \cap (F \times F)$ can be decided, in time $\mathcal{O}(2^{\mathcal{P}(|F|,|\mathcal{A}|)})$.

Note that the *T*, *D*, *F*, and *R* preorders are clearly well-defined. However, the *S* preorder is (provably) not well-defined. Nevertheless, our results do apply to possible-futures equivalence, as we shall see in Remark 2.

Lemma 7. If *P* is well-defined, then the relation \sqsubseteq_i over $F \times 2^F$ can be computed in time which is exponential in n and polynomial in i.

4.1 Encoding PQ Preorder into Modal Logic

Definition 9. For all $i \in \mathbb{N}_0$, $f \in F$, and $M \subseteq F$ we define the sets

 $\begin{array}{l} - \ \mathcal{F}(f,\sqsubseteq_i) = \{M \subseteq F \mid f \sqsubseteq_i M\} \\ - \ \mathcal{F}(\sqsubseteq_i,M) = \{f \in F \mid f \sqsubseteq_i M\}. \end{array}$

For all $f \in F$ and $k \in \mathbb{N}_0$ we define the formulae $\Phi_{f,k}$, $\Psi_{f,k}$, and $\Theta_{f,k}$ inductively as follows:

$$-\Phi_{f,0}=\Xi_f, \Psi_{f,0}=\Gamma_f$$

$$- \Theta_{f,k} = \Phi_{f,k} \wedge \Psi_{f,k}$$

- $\Phi_{f,k+1} = \Xi_f \wedge (\mathbf{AG} \bigvee_{f' \in F} \Theta_{f',k}) \wedge (\bigwedge_{f \stackrel{\mathfrak{Q}}{\to} f'} (\bigvee_{M \in \mathcal{F}(f', \sqsubseteq_k)} (\bigwedge_{f'' \in M} \langle\!\langle \alpha \rangle\!\rangle \Theta_{f'',k})))$
- $-\Psi_{f,k+1} = \Gamma_f \wedge (\mathbf{AG}\bigvee_{f'\in F} \Theta_{f',k}) \wedge \bigwedge_{\alpha\in\mathcal{A}_{\tau}} [\alpha](\bigvee_{f\subseteq M}\bigvee_{f'\in\mathcal{F}(\sqsubseteq_k,M)} \Theta_{f',k})$

The empty conjunction is equivalent to tt, and the empty disjunction to ff.

The $\mathcal{F}(\ldots)$ sets are effectively constructible in time exponential in n and polynomial in i (Lemma7), hence the $\Phi_{f,k}, \ldots$, formulae are effectively constructible too.

Theorem 4. Let g be an (arbitrary) process such that $\mathcal{A}(g) \subseteq \mathcal{A}$. Then for all $f \in F$ and $k \in \mathbb{N}_0$ we have the following:
- (a) $g \models \Phi_{f,0}$ iff $f \sqsubseteq_0 g$; further, $g \models \Phi_{f,k+1}$ iff $f \sqsubseteq_{k+1} g$ and for each $g \rightarrow^* g'$ there is $f' \in F$ such that $g' \sim_k f'$.
- (b) $g \models \Psi_{f,0}$ iff $g \sqsubseteq_0 f$; further, $g \models \Psi_{f,k+1}$ iff $g \sqsubseteq_{k+1} f$ and for each $g \to^* g'$ there is $f' \in F$ such that $g' \sim_k f'$.
- (c) $g \models \Theta_{f,0}$ iff $g \stackrel{\circ}{=}_0 f$; further, $g \models \Theta_{f,k+1}$ iff $f \sim_{k+1} g$ and for each $g \rightarrow^* g'$ there is $f' \in F$ such that $g' \sim_k f'$.

Corollary 2. Let g be an (arbitrary) process such that $\mathcal{A}(g) \subseteq \mathcal{A}$, and let $f \in F$. Then the following two conditions are equivalent:

(a) $g \sim f$ and for every $g \to^* g'$ there is some $f' \in F$ such that $g' \sim f'$. (b) $g \models \Theta_{f,n2^n} \wedge \operatorname{AG}(\bigvee_{f' \in F} \Theta_{f',n2^n})$.

Note that the size of the circuit representing the formula $\Theta_{f,n2^n} \wedge \mathbf{AG}(\bigvee_{f' \in F} \Theta_{f',n2^n})$ is exponential in n and can be constructed in exponential time.

Remark 2. As we already mentioned, the S preorder is not well-defined, because trace equivalence with a given finite-state process f is not expressible in modal logic (even monadic second order logic is (provably) not sufficiently powerful to express that a process can perform every trace over a given finite alphabet). Nevertheless, in our context it suffices to express the condition of *full trace equivalence* with f, which is achievable. So, full possible-futures equivalence with f is expressed by the formula $\Theta_{f,n2^n} \wedge AG(\bigvee_{f' \in F} \Theta_{f',n2^n})$ where for every $f' \in F$ we define $\Xi_{f'}$ and $\Gamma_{f'}$ to be the formula which expresses full trace equivalence with f'. This "trick" can be used also for other trace-like equivalences where the associated P is not well-defined.

5 Model Checking Lossy Channel Systems

In this section we show that the model checking of $\mathcal{L}(\mathbf{EX}_{\alpha}, \mathbf{EF}, \mathbf{EF}_{\tau}, \mathbf{EU}_{\alpha})$ formulae is decidable for lossy channel systems (LCS's). This result was inspired by [6] and can be seen as a natural extension of known results.

We refer to [1, 29] for motivations and definitions on LCS's. Here we only need to know that a *configuration* σ of a LCS *C* is a pair $\langle q, w \rangle$ of a control state *q* from some finite set *Q* and a finite word $w \in \Sigma^*$ describing the current contents of the channel (for simplicity we assume a single channel). Here $\Sigma = \{a, b, \ldots\}$ is a finite alphabet of messages. The behavior of *C* is given by a transition system T_C where steps $\sigma \to \sigma'$ describe how the configuration can evolve. In the rest of this section, we assume a fixed LCS *C*.

Saying that the system is *lossy* means that messages can be lost while they are in the channel. This is formally captured by introducing an ordering between configurations: we write $\langle q_1, w_1 \rangle \leq \langle q_2, w_2 \rangle$ when $q_1 = q_2$ and w_1 is a subword of w_2 (i.e. one can obtain w_1 by erasing some letters in w_2 , possibly all letters,

possibly none). Higman's lemma states that \leq is a well-quasi-ordering (a *wqo*), i.e. it is well-founded and any set of incomparable configurations is finite.

Losing messages in a configuration σ yields some σ' with $\sigma' \leq \sigma$. The crucial fact we shall use is that steps of LCS's are closed under losses:

Lemma 8 (see [1, 29]). If $\sigma \to \sigma'$ is a step of T_C , then for all configurations $\theta \ge \sigma$ and $\theta' \le \sigma'$, $\theta \to \theta'$ is a step of T_C too.

We are interested in sets of configurations denoted by some simple expressions. For a configuration σ we let $\uparrow \sigma$ denote the upward-closure of σ , i.e. the set $\{\theta \mid \sigma \leq \theta\}$. A *restricted set* is denoted by an expression ρ of the form $\uparrow \sigma - \uparrow \theta_1 - \cdots - \uparrow \theta_n$ (for some configurations $\theta_1, \ldots, \theta_n$). This denotes an upward-closure minus some restrictions (the $\uparrow \theta_i$'s).

An expression ρ is *trivial* if it denotes the empty set. Clearly $\uparrow \sigma - \uparrow \theta_1 - \cdots - \uparrow \theta_n$ is trivial iff $\theta_i \leq \sigma$ for some *i*. A *constrained set* is a finite union of restricted sets, denoted by an expression γ of the form $\rho_1 \vee \cdots \vee \rho_m$. Such an expression is *reduced* if no ρ_i is trivial. For a set *S* of configurations, $Pre(S) = \{\sigma \mid \exists \theta \in S, \sigma \to \theta\}$ is the set of (immediate) predecessors of configurations in *S*.

Lemma 9. Constrained sets are closed under intersection, complementation, and Pre. Furthermore, from reduced expressions γ , γ_1 and γ_2 , one can compute reduced expressions for $\gamma_1 \wedge \gamma_2$, $\neg \gamma$, and $Pre(\gamma)$.

We can now compute the set of configurations that satisfy an EU formula:

Lemma 10. Let S_1 and S_2 be two constrained sets. Then the set S of configurations that satisfy $S_1 \in U S_2$ is constrained too. Furthermore, from reduced expressions for S_1 and S_2 , one can compute a reduced expression for S.

By combining Lemma 9 and Lemma 10, we obtain the result we were aiming at:

Corollary 3. Let φ be a modal formula in $\mathcal{L}(\mathbf{EX}, \mathbf{EU})$. The set of configurations that satisfy φ is a constrained set, and one can compute a reduced expression for this set.

Theorem 5. The model checking problem for $\mathcal{L}(\mathbf{EX}_{\alpha}, \mathbf{EF}, \mathbf{EF}_{\tau}, \mathbf{EU}_{\alpha})$ formulae is decidable for lossy channel systems.

6 Applications

A Note on Semantic Quotients. Let $\mathcal{T} = (S, \rightarrow, \mathcal{A})$ be a transition system, $g \in S$, and ~ a process equivalence. Let $Reach(g) = \{s \in S \mid g \rightarrow^* s\}$. The ~-quotient of g is the process [g] of the transition system $(Reach(g)/\sim, \rightarrow, \mathcal{A})$ where $[s] \xrightarrow{\alpha} [t]$ iff there are $s', t' \in Reach(g)$ such that $s \sim s', t \sim t'$, and $s' \xrightarrow{\alpha} t'$.

For most (if not all) of the existing process equivalences we have that $s \sim [s]$ for every process s (see [17,18]). In general, the class of temporal properties

preserved under ~-quotients is larger than the class of ~-invariant properties [18]. Hence, ~-quotients are rather robust descriptions of the original systems. Some questions related to formal verification can be answered by examining the properties of ~-quotients, which is particularly advantageous if the ~-quotient is finite (so far, mainly the bisimilarity-quotients have been used for this purpose). This raises two natural problems:

- (a) Given a process g and an equivalence \sim , is the \sim -quotient of g finite?
- (b) Given a process *g*, an equivalence ~, and a finite-state process *f*, is *f* the ~-quotient of *g*?

The question (a) is known as *the strong regularity problem* (see, e.g., [16] where it is shown that strong regularity wrt. simulation equivalence is decidable for one-counter nets). For bisimulation-like equivalences, the question (a) coincides with the standard regularity problem.

Using the results of previous sections, the problem (b) is reducible to the model-checking problem with the logic $\mathcal{L}(\mathbf{EX}_{\alpha}, \mathbf{EF}, \mathbf{EF}_{\tau})$. Let $\mathcal{F} = (F, \rightarrow, \mathcal{A})$ be a finite state system and ~ an MTB or PQ equivalence. Further, let us assume that the states of \mathcal{F} are pairwise non-equivalent (this can be effectively checked). Consider the formula

$$\varrho_{f} \equiv \xi_{f} \land \bigwedge_{\substack{f' \in F}} \mathbf{EF} \ \xi_{f'} \land \bigwedge_{\substack{f' \xrightarrow{\alpha} \to f'' \\ (\text{in } \mathcal{F})}} \mathbf{EF} \ (\xi_{f'} \land \mathbf{EX}_{\alpha} \xi_{f''}) \land \bigwedge_{\substack{f' \xrightarrow{\alpha} \to f'' \\ (\text{in } \mathcal{F})}} \mathbf{AG} \ (\xi_{f'} \Rightarrow \mathbf{AX}_{\alpha} \neg \xi_{f''})$$

where ξ_f is the formula expressing full ~-equivalence with f. It is easy to see that for every process g s.t. $\mathcal{A}(g) \subseteq \mathcal{A}(f)$ we have that $g \models \varrho_f$ iff f is the ~-quotient of g.

Observe that if the problem (b) above is decidable for a given class of processes, then the problem (a) is semidecidable for this class. So, for all those models where model-checking with the logic $\mathcal{L}(\mathbf{EX}_{\alpha}, \mathbf{EF}, \mathbf{EF}_{\tau})$ is decidable we have that the positive subcase of the strong regularity problem is semidecidable due to rather generic reasons, while establishing the semidecidability of the negative subcase is a model-specific part of the problem.

Results for Concrete Process Classes. All of the so far presented results are applicable to those process classes where model-checking the relevant fragment of modal logic is decidable. In particular, model-checking $\mathcal{L}(\mathbf{EX}_{\alpha}, \mathbf{EF}, \mathbf{EF}_{\tau})$ is decidable for

- pushdown processes. In fact, this problem is **PSPACE**-complete [36]. Moreover, the complexity of the model-checking algorithm depends on the size of the circuit which represents a given formula (rather than on the size of the formula itself) [37];
- PA (and in fact also PAD) processes [24, 22]. The best known complexity upper bound for this problem in non-elementary.
- lossy channel systems (see Section 5). Here the model-checking problem is of nonprimitive recursive complexity.

Prom this we immediately obtain that the problem of full MTB-equivalence, where B is well-defined, is

- decidable in polynomial space for pushdown processes. For many concrete *MTB*-equivalences, this bound is optimal (for example, all bisimulation-like equivalences between pushdown processes and finite-state processes are **PSPACE**-hard [23]);
- decidable for PA and PAD processes;
- decidable for lossy channel systems. For most concrete *MTB*-equivalences, the problem is of nonprimitive recursive complexity (this can be easily derived using the results of [29]).

Similar results hold for PQ-equivalences where P is well-defined (for pushdown processes we obtain **EXPSPACE** upper complexity bound). Finally, the remarks about the problems (a),(b) of the previous paragraph also apply to the mentioned process classes.

References

- P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *I&C*, 127(2):91–101, 1996.
- [2] P.A. Abdulla, K. Čerāns, B. Jonsson, and Yih-Kuen Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *I&C*, 160(1–2):109–127, 2000.
- [3] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Decidability of bisimulation equivar lence for processes generating context-free languages. JACM, 40(3):653–682, 1993.
- [4] J.C.M. Baeten and R.J. van Glabbeek. Another look at abstraction in processalgebra. In *Proceedings of ICALP*'87, volume 267 of *LNCS*, pages 84–94. Springer, 1987.
- [5] A. Bouajjani. Languages, rewriting systems, and verification of infinite-state systems. In *Proceedings of ICALP'2001*, volume 2076 of *LNCS*, pages 24–39. Springer, 2001.
- [6] A. Bouajjani and R. Mayr. Model-checking lossy vector addition systems. In Proceedings of STACS'99, volume 1563 of LNCS, pages 323–333. Springer, 1999.
- [7] M.C. Browne, E.M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *TCS*, 59(1–2):115–131, 1988.
- [8] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 545–623. Elsevier, 2001.
- [9] J. Esparza and M. Nielsen. Decidability issues for Petri nets a survey. Journal of Information Processing and Cybernetics, 30(3):143–160, 1994.
- [10] A. Finkel and Ph. Schnoebelen. Well structured transition systems everywhere! *TCS*, 256(1–2):63–92, 2001.
- [11] Y. Hirshfeld and M. Jerrum. Bisimulation equivalence is decidable for normed process algebra. In *Proceedings of ICALP'99*, volume 1644 of *LNCS*, pages 412–421. Springer, 1999.
- [12] Y. Hirshfeld, M. Jerrum, and F. Moller. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *TCS*, 158(1–2):143–159, 1996.
- [13] Y. Hirshfeld, M. Jerrum, and F. Moller. A polynomial algorithm for deciding bisimulation equivalence of normed basic parallel processes. *MSCS*, 6(3):251–259, 1996.

- [14] P. Jančar. Undecidability of bisimilarity for Petri nets and some related problems. TCS, 148(2):281–301, 1995.
- [15] P. Jančar, A. Kučera, and R. Mayr. Deciding bisimulation-like equivalences with finite-state processes. *TCS*, 258(1–2):409–433, 2001.
- [16] P. Jančar, A. Kučera, and F. Moller. Simulation and bisimulation over onecounter processes. In *Proceedings of STACS'2000*, volume 1770 of *LNCS*, pages 334–345. Springer, 2000.
- [17] A. Kučera. On finite representations of infinite-state behaviours. *IPL*, 70(1):23–30, 1999.
- [18] A. Kučera and J. Esparza. A logical viewpoint on process-algebraic quotients. *JLC*, 13(6):863–880, 2003.
- [19] A. Kučera and P. Jančar. Equivalence-checking with infinite-state systems: Techniques and results. In *Proceedings of SOFSEM*'2002, volume 2540 of *LNCS*, pages 41–73. Springer, 2002.
- [20] A. Kučera and R. Mayr. A generic framework for checking semantic equivalences between pushdown automata and finite-state automata. In *Proceedings of IFIP TCS*'2004. Kluwer, 2004. To appear.
- [21] A. Kučera and Ph. Schnoebelen. A general approach to comparing infinite-state systems with their finite-state specifications. Technical report FIMU-RS-2004-05, Faculty of Informatics, Masaryk University, 2004.
- [22] D. Lugiez and Ph. Schnoebelen. The regular viewpoint on PA-processes. *TCS*, 274(1–2):89–115, 2002.
- [23] R. Mayr. On the complexity of bisimulation problems for pushdown automata. In *Proceedings of IFIP TCS'2000*, volume 1872 of *LNCS*, pages 474–488. Springer, 2000.
- [24] R. Mayr. Decidability of model checking with the temporal logic EF. *TCS*, 256(1–2):31–62, 2001.
- [25] R. Milner. Communication and Concurrency. Prentice-Hall, 1989.
- [26] M. Müller-Olm. Derivation of characteristic formulae. ENTCS, 18, 1998.
- [27] J. Parrow and P. Sjödin. Multiway synchronization verified with coupled simulation. In *Proceedings of CONCUR'92*, volume 630 of *LNCS*, pages 518–533. Springer, 1992.
- [28] J. Parrow and P. Sjödin. The complete axiomatization of cs-congruence. In Proceedings of STACS'94, volume 775 of LNCS, pages 557–568. Springer, 1994.
- [29] Ph. Schnoebelen. Verifying lossy channel systems has nonprimitive recursive complexity. *IPL*, 83(5):251–261, 2002.
- [30] G. Sénizergues. L(A)=L(B)? Decidability results from complete formal systems. TCS, 251(1–2):1–166, 2001.
- [31] J. Srba. Roadmap of infinite results. EATCS Bulletin, 78:163-175, 2002.
- [32] B. Steffen and A. Ingólfsdóttir. Characteristic formulae for processes with divergence. *I&C*, 110(1):149–163, 1994.
- [33] R.J. van Glabbeek. The linear time—branching time spectrum II: The semantics of sequential systems with silent moves. In *Proceedings of CONCUR '93*, volume 715 of *LNCS*, pages 66–81. Springer, 1993.
- [34] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *JACM*, 43(3):555–600, 1996.
- [35] M. Voorhoeve and S. Mauw. Impossible futures and determinism. *IPL*, 80(1):51–58, 2001.
- [36] I. Walukiewicz. Model checking CTL properties of pushdown systems. In Proceedings of FST&TCS'2000, volume 1974 of LNCS, pages 127–138. Springer, 2000.
- [37] I. Walukiewicz. Private communication, September 2003.

Model Checking Timed Automata with One or Two Clocks

F. Laroussinie¹, N. Markey^{1,2}, and Ph. Schnoebelen¹

 ¹ Lab. Spécification & Vérification ENS de Cachan & CNRS UMR 8643
61, av. Pdt. Wilson, 94235 Cachan Cedex, France {fl,markey,phs}@lsv.ens-cachan.fr
² Département d'Informatique – CP 212 Université Libre de Bruxelles Bd du Triomphe, 1050 Bruxelles, Belgique nmarkey@ulb.ac.be

Abstract. In this paper, we study model checking of timed automata (TAs), and more precisely we aim at finding efficient model checking for subclasses of TAs. For this, we consider model checking TCTL and $TCTL_{\leq,\geq}$ over TAs with one clock or two clocks.

First we show that the reachability problem is NLOGSPACE-complete for one clock TAs (*i.e. as* complex as reachability in classical graphs) and we give a polynomial time algorithm for model checking $TCTL_{\leq,\geq}$ over this class of TAs. Secondly we show that model checking becomes PSPACE-complete for full *TCTL* over one clock TAs. We also show that model checking *CTL* (without any timing constraint) over two clock TAs is PSPACE-complete and that reachability is NP-hard.

1 Introduction

Model checking is widely used for the design and debugging of critical reactive systems [Eme90,CGP99]. During the last decade, it has been extended to *real-time systems*, where quantitative information about time is required.

Timed Models. Real-time model checking has been mostly studied and developed in the framework of Alur and Dill's *Timed Automata* (TAs) [ACD93,AD94], *i.e.* automata extended with *clocks* that progress synchronously with time. There now exists a large body of theoretical knowledge and practical experience for this class of systems. It is agreed that their main drawback is the complexity blowup induced by timing constraints: most verification problems are at least PSPACE-hard for Timed Automata [Alu91,CY92,ACD93,AL02].

Real-time automata are TAs with a unique clock which is reset after every transition. This subclass has been mostly studied from the language theory point of view [Dim00], but it is also considered in [HJ96] for modeling real-time systems. Clearly this subclass is less expressive than classical TAs with an arbitrary

number of clocks but still it is natural and convenient for describing behavior of simply timed systems. For example, it may be useful to model systems where timing constraints are local, *i.e.* depend only of the time elapsed since the last transition. The use of a real valued clock offers a convenient and abstract concept of time. Moreover such kinds of restricted TAs are more natural and more expressive than models based on discrete Kripke Structures where some durations are associated with transitions (see for example Timed Transition Graphs [CC95] or Durational Kripke Structures [LMS02]).

Timed Specifications. In order to express timing aspects of computations, we consider extensions of the classical temporal logic *CTL*. The idea is to use timing constraints tagging temporal modalities [AH92]. For example, the formula $\mathsf{EF}_{<10}$ A states that it is possible to reach a state verifying A ("EF A") in less than 10 time units. Timing constraints can have three main forms: " $\leq c$ " and " $\geq c$ " set a lower or upper bound for durations, while "= c" requires a precise value. *TCTL* is the extension of *CTL* with all three types of constraints, while *TCTL*_{\leq ,>} is the fragment of *TCTL* where the "=c" constraints are forbidden.

Our Contribution. In this paper, we aim at finding subclasses of Timed Automata that admit efficient model checking algorithms. For this purpose we consider one clock TAs (1C-TAs) which extend real-time automata because the clock is not required to be reset after each transition. First we show that reachability problem is NLOGSPACE-complete for 1C-TAs (*i.e.* as efficient as reachability in classical graphs) and we give a polynomial time algorithm for model checking $TCTL_{\leq,\geq}$ over 1C-TAs. These results are surprising because adding simple timing constraints induces generally a complexity blowup. Note efficient model checking $TCTL_{\leq,\geq}$ over 1C-TAs requires to use an *ad-hoc* algorithm: the classical region graph technique or the symbolic algorithms based on DBMs [Dil90] are not polynomial over this subclass.

Secondly we show that model checking becomes PSPACE-complete for full *TCTL* over 1C-TAs. Then we address the case of TAs with two clocks (2C-TAs), since it is well known that three clocks lead to PSPACE-hardness for reachability [CY92]. We show that model checking *CTL* (without any timing constraints) over 2C-TAs is already PSPACE-complete and that reachability is NP-hard.

These results emphasize the good properties of 1C-TAs and real-time automata, leading to efficient timed model checking.

Related Work. Quantitative logics *for Timed Automata* are now well-known and many results are available regarding their expressive power, or the satisfiability and model checking problems [AH94,ACD93,AH93,AFH96,Hen98]. That exact durations may induce harder model checking complexity was already observed in the case of *LTL* and Timed Automata [AFH96]. Complexity of timed model checking is considered in [CY92] where it is shown that three clocks are sufficient to have PSPACE-hardness for the reachability problem. In [AL02], model

checking is studied for several timed modal logics. In [ACH94] the expressive power of clocks in TAs is studied from the language theory point of view.

2 Timed Automata

Let \mathbb{N} and \mathbb{R} denote the sets of natural and non-negative real numbers, respectively. Let \mathcal{C} be a set of real valued clocks. We use $\mathcal{B}(\mathcal{C})$ to denote the set of boolean expressions over atomic formulae of the form ${}^1 x \sim k$ with $x \in \mathcal{C}, k \in \mathbb{N}$, and $\sim \in \{<, \leq, >, \geq, =\}$. Constraints of $\mathcal{B}(\mathcal{C})$ are interpreted over *valuations* for \mathcal{C} clocks, that are functions from \mathcal{C} to \mathbb{R} . The set of valuations is denoted by $\mathbb{R}^{\mathcal{C}}$. For every $v \in \mathbb{R}^{\mathcal{C}}$ and $d \in \mathbb{R}$, we use v + d to denote the time assignment which maps each clock $x \in \mathcal{C}$ to the value v(x) + d. For every $r \subseteq \mathcal{C}$, we write $v[r \leftarrow 0]$ for the valuation which maps each clock in r to the value 0 and agrees with v over $\mathcal{C} \setminus r$. Let AP be a set of atomic propositions.

Definition 2.1. A timed automaton (*TA*) is a 6-tuple $A = \langle Q_A, C, q_{init}, \rightarrow_A,$ $\ln v_A, l_A \rangle$ where Q_A is a finite set of control states, C is a finite set of clocks and $q_{init} \in Q_A$ is the initial state. The set $\rightarrow_A \subseteq Q_A \times \mathcal{B}(C) \times 2^C \times Q_A$ is a finite set of action transitions: for $(q, g, r, q') \in \rightarrow_A$, g is the enabling condition (or guard) of the transition and r is the set of clocks to be reset with the transition (we write $q \xrightarrow{g,r}_A q'$). $\ln v_A : Q_A \to \mathcal{B}(C)$ assigns a constraint, called an invariant, to any control state. Finally $l_A : Q_A \to 2^{AP}$ labels every control state with a subset of AP.

A state (or configuration) of a timed automaton A is a pair (q, v), where $q \in Q_A$ is the current control state and $v \in \mathbb{R}^C$ is the current clock valuation. The initial state of A is (q_{init}, v_0) where v_0 is the valuation mapping all clocks in C to 0.

There are two kinds of transition. From (q, v), it is possible to perform the *action transition* $q \xrightarrow{g,r}_A q'$ if $v \models g$ and $v[r \leftarrow 0] \models \ln v_A(q')$ and then the new configuration is $(q', v[r \leftarrow 0])$. It is also possible to let time elapsing, and reach (q, v + t) for some $t \in \mathbb{R}$ whenever the invariant is satisfied. Formally the semantics of a TA A is given by a Timed Transition System (TTS) $\mathcal{T}_A = (S, s_{\text{init}}, \rightarrow_{\mathcal{T}_A}, l)$ where:

$$-S = \{(q, v) \mid q \in Q_A \text{ and } v \in \mathbb{R}^{\mathcal{C}} \text{ s.t. } v \models \mathsf{Inv}_A(q) \} \text{ and } s_{\mathsf{init}} = (q_{\mathsf{init}}, v_0).$$

- → $\mathcal{T}_{\mathcal{A}} \subseteq S \times S$ and we have $(q, v) \rightarrow \mathcal{T}_{\mathcal{A}}(q', v')$ iff • either q' = q, v' = v + t and $v + t' \models \mathsf{Inv}_{\mathcal{A}}(q)$ for any $t' \leq t$ — we write $(q, v) \xrightarrow{\delta(t)} (q, v + t)$ —,
- or $\exists q \xrightarrow{g,r}_A q'$ and $v \models g, v' = v[r \leftarrow 0]$ and $v' \models \mathsf{Inv}_A(q')$ we write $(q, v) \rightarrow_a (q', v')$.
- $-l: S \rightarrow 2^{AP}$ labels every state (q, v) with the subset of $AP \ l_A(q)$.

¹ Considering diagonal constraints $x - y \sim k$ does not matter for the complexity.

An execution of A is an infinite path in $\mathcal{T}_{\mathcal{A}}$. Let s = (q, v) be an A-configuration. An execution ρ from s can be described as an infinite sequence $s = s_0 \xrightarrow{\delta(t_0)} \rightarrow_a s_1 \xrightarrow{\delta(t_1)} \rightarrow_a \cdots$ for some $t_i \in \mathbb{R}$. Such an execution ρ goes through any configuration s' reachable from some s_i by a delay transition of duration $t \in [0; t_i]$ — we write $s' \in \rho$. Let Exec(s) be the set of all executions from s.

The standard notions of prefix, suffix and subrun apply for paths in TTS. Given $\rho \in \text{Exec}(s)$, any finite prefix σ leading to a configuration s' (denoted $s \stackrel{\sigma}{\mapsto} s'$) has a *duration*, Time $(s \stackrel{\sigma}{\mapsto} s')$, defined as the sum of all delays along σ . Let Pref (ρ) be the set of all prefixes of ρ .

Given $\rho \in \text{Exec}(s)$ and $s', s'' \in \rho$, we say that s' precedes strictly s'' along ρ (written $s' <_{\rho} s''$) iff there exists a finite subrun σ in ρ s.t. $s' \xrightarrow{\sigma} s''$ and σ contains at least one non null delay transition or one action transition (*i.e.* σ is not reduced to $\xrightarrow{\delta(0)}$). Note that a configuration may have several occurrences along ρ and then it may be that $s <_{\rho} s$ or $s <_{\rho} s'$ and $s' <_{\rho} s$.

along ρ and then it may be that $s <_{\rho} s$ or $s <_{\rho} s'$ and $s' <_{\rho} s$. The size of a TA is $|Q_A| + |\mathcal{C}| + \sum_{(q,g,r,q') \in \rightarrow_A} |g| + \sum_q |\ln v_A(q)|$ where the size of a constraint is its length (constants are encoded in binary). We use $|\rightarrow_A|$ to denote the number of transitions in A.

3 Timed CTL

TCTL is the quantitative extension of *CTL* where temporal modalities are subscripted with constraints on duration [ACD93]. Formulae are interpreted over TTS states.

Definition 3.1 (Syntax of TCTL). *TCTL formulae are given by the following grammar:*

$$\varphi, \psi ::= P_1 \mid P_2 \mid \ldots \mid \neg \varphi \mid \varphi \land \psi \mid \mathsf{E}\varphi \mathsf{U}_{\sim c} \psi \mid \mathsf{A}\varphi \mathsf{U}_{\sim c} \psi$$

where ~ can be any comparator in $\{<, \leq, =, \geq, >\}$, c any natural number and $P_i \in AP$.

Standard abbreviations include $\top, \perp, \varphi \lor \psi, \varphi \Rightarrow \psi, \ldots$ as well as $\mathsf{EF}_{\sim c} \varphi$ (for $\mathsf{ETU}_{\sim c} \varphi$), $\mathsf{AF}_{\sim c} \varphi$ (for $\mathsf{ATU}_{\sim c} \varphi$), $\mathsf{EG}_{\sim c} \varphi$ (for $\neg \mathsf{AF}_{\sim c} \neg \varphi$) and $\mathsf{AG}_{\sim c} \varphi$ (for $\neg \mathsf{EF}_{\sim c} \neg \varphi$). Further, the modalities U, F and G without subscripts are shorthand for $\mathsf{U}_{\geq 0}$, $\mathsf{F}_{\geq 0}$ and $\mathsf{G}_{\geq 0}$. The size $|\varphi|$ of a formula φ is defined in the standard way, with constants written in binary notation.

Definition 3.2 (Semantics of TCTL). The following clauses define when a state *s* of some TTS $T = \langle S, s_{init}, \rightarrow, l \rangle$ satisfies a TCTL formula φ , written $s \models \varphi$, by induction over the structure of φ (semantics of boolean operators is omitted).

$$\begin{split} s &\models \mathsf{E}\varphi \mathsf{U}_{\sim c} \ \psi \ \textit{iff} \ \exists \ \rho \in \textit{Exec}(s) \ \textit{with} \ \rho = \sigma \cdot \rho' \ \textit{and} \ s \stackrel{\sigma}{\mapsto} s' \ \textit{s.t.} \\ & \mathsf{Time}(s \stackrel{\sigma}{\mapsto} s') \sim c \ , \ s' \models \psi \ \textit{and} \ \forall \ s'' <_{\rho} s', \ s'' \models \varphi \\ s \models \mathsf{A}\varphi \mathsf{U}_{\sim c} \ \psi \ \textit{iff} \ \forall \rho \in \textit{Exec}(s), \exists \sigma \in \textit{Pref}(\rho), \ \textit{s.t.} \ s \stackrel{\sigma}{\mapsto} s', \\ & \mathsf{Time}(s \stackrel{\sigma}{\mapsto} s') \sim c, \ s' \models \psi \ \textit{and} \ \forall \ s'' <_{\rho} s', \ s'' \models \varphi \end{split}$$

Thus, in $E\varphi U_{\sim c} \psi$, the classical until is extended by requiring that ψ be satisfied within a duration (from the current state) verifying the constraint " $\sim c$ ".

Given a TA $A = \langle Q, C, q_{\text{init}}, \rightarrow_A, \text{Inv}_A, l_A \rangle$ and a *TCTL* formula φ , we write $A \models \varphi$ when $s_{\text{init}} \models \varphi$.

4 Complexity of Timed Model Checking

Given a TA A, the TTS $\mathcal{T}_{\mathcal{A}}$ may have an infinite number of states and then standard model checking techniques cannot be applied directly. Indeed the decidability of verification problems over TAs is based on the region graph technique: The infinite state space of configurations is partitioned in a finite number of *regions* (equivalence classes of a relation over valuations) which have the "same behavior" w.r.t. the property to be checked, then a standard model checking algorithm can be applied over this finite abstraction. The region graph mainly depends on the number of clocks and the constants occurring in the guard. One of the main drawbacks of timed model checking is that the size of the region graph is exponential in the number of clocks and the (encoding of) constants. Several data-structures have been proposed to verify non-trivial timed systems (for ex. DBM see [Dil90,Bou04]).

Reachability problem of timed automata is known to be PSPACE-complete [AH94]. In [CY92], reachability in TA is shown to be PSPACE-complete even when the number of clocks is 3 or when the constants occurring in the guard belong to $\{0,1\}$.

For *TCTL*, model checking is PSPACE-complete [ACD93]. And it is EXP-TIME-complete for many variants of timed μ -calculus [AL02]; Checking timed bisimilarity is also an EXPTIME-complete problem. Note that all these results hold for a \mathbb{R} or \mathbb{N} as time domain and these results still hold when considering a parallel composition of TAs instead of a single one [AL02].

In this paper, we consider two subclasses of TAs whose complexity for timed verification is not known: we will study TAs with one clock (1C-TAs) or two clocks (2C-TAs). Clearly these subclasses are more expressive than real-time automata where the unique clock is reset after any transition and than extensions of Kripke structures with integer durations.

We will assume that in 1C-TAs, the guards are given by two constants defining the minimal (resp. maximal) value for x to perform the transition: it is always possible to reduce, in polynomial time, any 1C-TA to an equivalent automaton verifying such a property.

5 Model Checking One Clock Timed Automata

For a 1C-TA, a valuation is just a real value: the time assignment associated with the automaton clock x. First we consider the reachability problem: "Given a TA and a control state q, is it possible to reach a configuration (q, v) from the initial state?"

Proposition 5.1. Reachability in 1C-TAs is NLOGSPACE-complete.

Proof. The NLOGSPACE-hardness comes from complexity of reachability in classical graphs. Now we give a NLGOSPACE algorithm. A 1C-TA configuration is a control state and a value for the clock x. It is sufficient to consider only the integer value of x and to know if the fractional part is zero or not, but the integer value cannot be stored directly in a logarithmic space algorithm and we have to use a more concise encoding.

Let A be a 1C-TA. Let **B** be the set of integer values used in the guards and zero. We use b_0, b_1, \ldots, b_k to range over **B** and assume $0 = b_0 < b_1 < \cdots$ and $|\mathbf{B}| = k + 1$. The set **B** defines a set $\mathcal{I}_{\mathbf{B}}$ of 2(k + 1) intervals $\lambda_0, \lambda_1, \ldots$ with $\lambda_0 \stackrel{\text{def}}{=} [b_0; b_0], \lambda_1 \stackrel{\text{def}}{=} (b_0; b_1), \lambda_2 \stackrel{\text{def}}{=} [b_1; b_1], \cdots, \lambda_{2k+1} \stackrel{\text{def}}{=} (b_k, \infty)$. We will encode the configuration (q, x) by the pair (q, n(x)) s.t. $x \in \lambda_{n(x)}$. Since $k \leq 2 \cdot | \rightarrow_A |$, it is possible to store n(x) in logarithmic space.

First the algorithm counts the number of different constants in guards of A. This is done by verifying that the constants occurring in the *i*-th transition are different from the constants used in the *j*-th transition with j < i (this test is done by enumerating each bit of the constant c to be checked and verify the equivalence, it requires a space in $O(\log(\log(c)))$).

Then given a pair (q, n), the algorithm non-deterministically guesses another (q', n') and verifies that $(q', \lambda_{n'})$ is reachable from (q, λ_n) , *i.e.* either q = q' and n' = n + 1 (this is a delay transition), or there exists a transition $q \stackrel{g,r}{\to} q'$ s.t. g is satisfied by any value in λ_n and n' = n (resp. n' = 0) if $r = \emptyset$ (resp. $r = \{x\}$). Assume $g = m_1 \le x \le m_2$, then checking $\lambda_n \models g$ can be done by counting the number n_1 of different constants less than m_1 and the number n_2 of those greater than m_2 . Finally $\lambda_n \models g$ iff $\frac{n}{2} \ge n_1$ and $\frac{n}{2} \le k - n_2$ (resp. $\frac{n-1}{2} \ge n_1$ and $\frac{n+1}{2} \le k - n_2$) if n is even (resp. n is odd). These operations requires only a logarithmic space.

This result entails that analysing a 1C-TA is not more complex than analysing an untimed graph from the complexity theory. After this positive result, we now consider model checking for 1C-TA and $TCTL_{\leq,\geq}$:

Theorem 5.2. Model checking $TCTL_{\leq,\geq}$ over 1C-TAs is P-complete.

Proof. P-hardness follows from the case of *CTL* model checking. We present a polynomial algorithm to construct, for any state q and subformula ξ of Φ , an union of intervals $\mathsf{Sat}[q,\xi]$ over \mathbb{R} containing the valuations for x s.t. $x \in \mathsf{Sat}[q,\xi]$ iff $(q,x) \models \xi$. Assume $\mathsf{Sat}[q,\xi] = \bigcup_{j=1,\dots,k} \langle \alpha_j, \beta_j \rangle$ with $\langle \in \{[, \{\} \text{ and } \rangle \in \{], \}\}$; We will see that it is sufficient to consider $\alpha_j, \beta_j \in \mathbb{N} \cup \{\infty\}$. We choose $\alpha_j < \beta_j$

and $\beta_j < \alpha_{j+1}$ if $j+1 \le k$ in order to keep its size (*i.e.* the number of intervals) small; Indeed we will show that $|\mathsf{Sat}[q,\xi]| \le 2 \cdot |\xi| \cdot |\to_A|$. We denote by $\mathsf{Cst}_A \subseteq \mathbb{N} \cup \{\infty\}$ the set of all constants occurring in *A* (either in guards or in invariants) plus 0.

We only present here the labeling procedure for the modality $E\varphi U_{\leq c} \psi$: the case of boolean operators and atomic propositions is straightforward and the procedures for other modalities are given in Appendix A.

Assume $\xi = \mathsf{E}\varphi \mathsf{U}_{\leq c} \psi$. Assume also that $\mathsf{Sat}[q, \varphi]$ and $\mathsf{Sat}[q, \psi]$ have been already constructed. In order to compute $\mathsf{Sat}[q, \xi]$, we build a (finite) graph $G = (V_G, \rightarrow_G, l_G)$ where every node $v \in V_G$ corresponds to a set of configurations (q, λ) where λ is an interval over \mathbb{R} s.t. (1) these configurations verify either ψ or $\varphi \land \neg \psi$, (2) for any guard g in an A-transition, $\lambda \models g$ or $\lambda \models \neg g$. This last requirement implies that the same sequences of action transitions are enabled from any configuration of (q, λ) .

Every *G*-transition will correspond to an action transition of *A* or an abstract delay transition (leading to another node with different properties): *G* can be seen as a kind of region graph. The definition of intervals λ depends on $\mathsf{Sat}[q, \varphi]$ and $\mathsf{Sat}[q, \psi]$ and also on guards of *A*. Let \mathbb{B} be the finite set $\mathsf{Cst}_A \cup \{\alpha_j, \beta_j \mid \exists q \in Q_A \text{ s.t. } \langle \alpha_j; \beta_j \rangle \in \mathsf{Sat}[q, \varphi] \cup \mathsf{Sat}[q, \psi]\}$. We enumerate \mathbb{B} as b_0, b_1, \ldots with $b_i < b_{i+1}$. We define V_G as the pairs (q, λ) where (1) λ is of the form $[b_i; b_i]$ or $(b_i; b_{i+1})$, and (2) we have $\lambda \subseteq \mathsf{Sat}[q, \psi]$ or $\lambda \subseteq \mathsf{Sat}[q, \varphi] \cap \mathsf{Sat}[q, \psi]$. The *G*-transitions are:

- actions: $(q, \lambda) \rightarrow_G (q', \lambda')$ if there exists $q \xrightarrow{g,r} q'$ in A such that $\lambda \models g, \lambda' = \lambda$ (resp. $\lambda' = [0; 0]$) if $r = \emptyset$ (resp. $r = \{x\}$), and $\lambda' \models \mathsf{Inv}_A(q')$.
- abstract delays: $(q, \lambda) \rightarrow_G (q, \operatorname{Succ}(\lambda))$ if $\operatorname{Succ}(\lambda) \models \operatorname{Inv}_A(q)$, where Succ is the function: $\operatorname{Succ}([b_i, b_i]) = (b_i; b_{i+1})$ and $\operatorname{Succ}((b_i; b_{i+1})) = [b_{i+1}; b_{i+1}]$ if $b_{i+1} < \infty$ and $\operatorname{Succ}((b_i; \infty)) = (b_i; \infty)$ otherwise.

Note that $|G| \leq (Q_A \cdot 2 \cdot |\mathbb{B}|) \cdot (2 + | \rightarrow_A |)$. We can now restrict G to the nodes satisfying $\mathsf{E}\varphi \mathsf{U}\psi$ by a standard algorithm and then clearly the nodes in V_G represent all A configurations satisfying $\mathsf{E}\varphi \mathsf{U}\psi$. We now have to see when there exists a path leading to a ψ -state and being short enough (*i.e.* $\leq c$) to witness ξ . For this we can compute for any node $(q, \lambda) \in V_G$ a *duration function* $\delta_{q,\lambda}^{\psi}: \lambda \to \mathbb{R}$ s.t. $\delta_{q,\lambda}^{\psi}(t)$ is the duration of a shortest path from (q, t) to some state verifying ψ (along a path satisfying φ). The crucial point is that such a duration function function over λ has a special structure: it is first constant and then decreases with the slope -1. The constant part corresponds to configurations for which a shortest path starts by a sequence of action transitions where the clock is reset at least once before any time elapsing (and clearly this also holds for previous positions in λ), and the decreasing part corresponds to positions from which a delay transition occurs before reseting x along a shortest path. These functions can easily be encoded as pairs (c_1, c_2) with $c_1 \geq c_2$, with the following meaning:

$$\begin{split} \delta^{\psi}_{q,[b_i;b_i]}(t) \stackrel{\text{def}}{=} c_1 \\ \delta^{\psi}_{q,(b_i;b_{i+1})}(t) \stackrel{\text{def}}{=} \begin{cases} c_1 & \text{if } b_i < t \le b_{i+1} - (c_1 - c_2) \\ c_2 - (t - b_{i+1}) & \text{if } b_{i+1} - (c_1 - c_2) < t < b_{i+1} \end{cases} \end{split}$$

TEAM LING

Of course, it is also possible to have a pure constant function over λ (then $c_1 = c_2$) or a pure decreasing function (then $c_1 = c_2 + (b_{i+1} - b_i)$). See Figure 1 for more intuition.



Fig. 1. Example of duration functions

The structure of the duration functions allows us to compute them by adapting the Bellman-Ford algorithm for single source shortest path over G. This algorithm is given in Appendix A. The idea is to compute the $\delta_{q,\lambda}^{\psi}$'s by successive approximations. Consider a shortest path (SP) π in $\mathcal{T}_{\mathcal{A}}$ starting from (q_0, x_0) , leading to a state verifying ψ with intermediary states satisfying φ . The path π can be described as a sequence of $\stackrel{\delta(t_i)}{\longrightarrow}_a$. Such a path in $\mathcal{T}_{\mathcal{A}}$ is associated with a path in G where the delay transitions $\stackrel{\delta(t_i)}{\longrightarrow}$ are replaced by a sequence of abstract delay transitions. Clearly along a SP, a node (q, λ) occurs at most once: given a configuration (q, x) with $x \in \lambda$, either a SP starts as the previous positions x' < x in λ and it starts by action transitions that can be performed from (q, x), or the SP starts by delaying until $\operatorname{Succ}(\lambda)$ and in both cases it is not necessary to come back to (q, λ) later. Assume the size of a SP in G is k, then k is bounded by $|V_G| + 1$ and then it is discovered after the k-th step of the algorithm.

Once the $\delta_{q,\lambda}^{\psi}$'s have been computed, it remains to see which intervals or part of intervals contain positions whose distance to ψ is less than *c*. This step may lead to cut an interval in two parts (still at an integer point) and add new constants in **B**; in Appendix A we show that the size of $\mathsf{Sat}[q,\xi]$ is bounded by $|\mathsf{Sat}[q,\psi]| + (|\mathsf{Sat}[q,\varphi]| + 2 \cdot |\to_A|)$ and the number of new constants in **B** is bounded by $|\to_A|$.

From the previous procedure and those in Appendix A, we can deduce that $|\operatorname{Sat}[q,\varphi]| \leq 2 \cdot |\varphi| \cdot |\to_A|$. This entails that the most complex procedure $(E_-\mathsf{U}_{<-})$ runs in $O(|\xi|^2 \cdot |Q_A|^2 \cdot |\to_A|^3)$. This globally provides a complexity of $O(|\Phi|^3 \cdot |Q_A|^2 \cdot |\to_A|^3)$ for the full labeling procedure. More precisely we could show that the algorithm is in $O(|\Phi| \cdot |Q_A|^2 \cdot |\to_A| \cdot (\operatorname{Cst}_A + N_{\Phi}^{\sim c})^2)$ where $N_{\Phi}^{\sim c}$ is the number of Φ subformulae of the form $\operatorname{EU}_{\sim c}$ or $\operatorname{AU}_{\sim c}$.

When considering exact durations in subscripts, model checking becomes PSPACE-hard, *i.e.* as hard as model checking TAs with several clocks:

Theorem 5.3. Model checking TCTL on 1C-TAs is PSPACE-complete.

Proof. Membership in PSPACE follows from the general result for TAs [ACD93]. PSPACE-hardness is shown by reducing QBF instance to a model checking problem over 1C-TA.

Consider a QBF instance $\Phi \stackrel{\text{def}}{=} Q_0 p_0 Q_1 p_1 \dots Q_{n-1} p_{n-1} \cdot \varphi$: $Q_i \in \{\exists, \forall\}$, any p_i is a boolean variable for $i = 0, \dots, n-1$, and φ is a propositional formula over the p_i 's.



Fig. 2. 1C-TA A_{Φ} associated with QBP instance Φ

To reduce the QBF instance Φ to a model checking problem, we consider the 1C-TA A_{Φ} depicted in Figure 2 and the formulae $\overline{\Phi}_i$ with i = 0, ..., n defined as:

$$0 \leq i < n : \overline{\Phi_i} \stackrel{\text{def}}{=} \begin{cases} \mathsf{EF}_{=2^i} \left((p_i \vee \overline{p_i}) \wedge \overline{\Phi_{i+1}} \right) \text{ if } Q_i = \exists AF_{=2^i} \left((p_i \vee \overline{p_i}) \wedge \overline{\Phi_{i+1}} \right) \text{ if } Q_i = \forall \overline{\Phi_n} \stackrel{\text{def}}{=} \varphi[p_i \leftarrow \mathsf{EF}_{=2^n-1} b_i^\top] \end{cases}$$

Now we show that Φ is valid iff $(q_0, 0) \models \overline{\Phi_0}$. Indeed, interpreting $\overline{\Phi_0}$ over $(q_0, 0)$ makes that every formula $\overline{\Phi_i}$ with $i = 1, \ldots, n$ is interpreted over some configurations in a set S_i located at duration $\sum_{j < i} 2^j$ from $(q_0, 0)$. More precisely S_i is composed by (p_{i-1}, l) and $(\overline{p_{i-1}}, l)$ with $l \in \{1, \ldots, 2^{i-1}\}$. A configuration in S_i can be seen as a boolean valuation for p_0, \ldots, p_{i-1} : The truth value of p_{i-1} is T iff the control state is p_{i-1} and the value of p_k (k < i - 1) is given by the k-th bit of the binary encoding of l - 1. Moreover this valuation is preserved in the two possible successor configurations in S_{i+1} at duration 2^i from the current position. The alternation of existential EF and AF allows to simulate the alternation of quantifiers over the p_i 's in Φ .

Finally Φ_n is interpreted over configurations of S_n which define valuations for p_0, \ldots, p_{n-1} . The configurations of the form (p_{n-1}, l) (resp. $(\overline{p_{n-1}}, l)$) with $l \in \{1, \ldots, 2^{n-1}\}$ are located at distance $0, \ldots, 2^{n-1}-1$ (resp. $2^{n-1}, \ldots, 2^n-1$) to q_F . Consider such a configuration (p_{n-1}, l) and assume $(p_{n-1}, l) \models \mathsf{EF}_{=2^n-1} b_k^\top$: Reaching $(q_F, 0)$ takes $2^{n-1} - l$, it remains to spend $2^{n-1} + l - 1$ in the loop $b_k^\perp b_k^\top \cdots$ and clearly b_k^\top holds after this duration iff the k-th bit of l-1 is 1. \Box

Note that the automaton depicted in Figure 2 is a real-time automaton (x is reset after every transition) and then we can deduce the following corollary:

Corollary 5.4.

- Reachability in real-time automata is NLOGSPACE-complete.
- Model checking TCTL_{<,>} over real-time automata is P-complete.
- Model checking TCTL over real-time automata is PSPACE-complete.

6 Model Checking Two Clocks Timed Automata

When a timed automaton has two clocks, there is a complexity blow-up for model checking. First we have the following result for reachability:

Proposition 6.1. Reachability problem in 2C-TAs is NP-hard.

Proof. This follows from a simple encoding of the SUBSET-SUM problem [GJ79, p. 223]: assume we are given a set $\{a_1, \ldots, a_p\}$ of integers and a *goal* b, one asks whether there exists a subset $J \subseteq \{1, \ldots, p\}$ s.t. $\sum_{j \in J} a_j = b$. This problem is known to be NP-complete.

This problem is obviously equivalent to the reachability problem for state G in the automaton shown on figure 3.



Fig. 3. Encoding of SUBSET-SUM in a 2C-TA

This complexity blow-up compared to the one clock case increases when considering model-checking:

Theorem 6.2. The model checking problems for CTL, $TCTL_{\leq,\geq}$ or TCTL on 2C-TAs are PSPACE-complete.

Proof. The PSPACE-membership comes from PSPACE model checking algorithm for *TCTL* over classical TAs. It is sufficient to show PSPACE-hardness for the *CTL* case. Let $\Phi \stackrel{\text{def}}{=} O_0 p_0, O_1 p_1 \cdots O_{n-1} p_{n-1} \cdot \varphi$ be a QBF instance $(O_i \in \{\exists, \forall\} \text{ and } \varphi \text{ is boolean formula over the } p_i$'s). Consider the 2C-TA depicted in Figure 4.



Fig. 4. The 2C-TA A_{Φ} associated with the QBF instance Φ

Let $\overline{\Phi}$ be the following *CTL* formula:

$$\overline{\Phi} \stackrel{\text{def}}{=} \left(O_0 q_0 \mathsf{U} \left(q_1 \land (O_1 q_1 \mathsf{U} (q_2 \land \dots \mathsf{U} (q_n \land \overline{\varphi}))) \right) \right)$$

with $\overline{\varphi} \stackrel{\text{def}}{=} \varphi[p_i \leftarrow \mathsf{EF}p_i]$. A path from q_0 to q_n defines a boolean valuation for the p_i 's: performing the transition $q_i \xrightarrow{x=2^i, x:=0} q_{i+1}$ (resp. $q_i \xrightarrow{x=0} q_{i+1}$) assigns \top (resp. \bot) to p_i . And in the configuration $(q_n, 0, v_y)$, the valuation is encoded in the value v_y (the total amount of time used to reach q_n). Then the branch $q_n \rightarrow s_i \rightarrow s_{i,1} \cdots$ allows us to check the value of the *i*-th bit of v_y , that is exactly the truth value of p_i .

Note that this last result is proved for a very simple subclass: the automaton used in proof of Theorem 6.2 has a clock which is reset after each transition. Despite this, model checking (untimed) *CTL* leads to PSPACE-hardness.

7 Conclusion

Figure 5 gives an overview of the results presented in the paper and a comparison with the results for classical Timed Automata. The main results concern oneclock automata. First the reachability problem in 1C-TAs is as efficient as the reachability in classical graphs. Moreover model checking can be done efficiently if the property is expressed with $TCTL_{\leq,\geq}$ logic. This result is surprising because usually, in *TCTL* model checking, the timing constraints are handled by adding a new clock in the system and we also have seen that any model checking problem, even for the untimed *CTL*, is PSPACE-hard over simple 2C-TAs. Moreover note that the efficiency requires an *ad hoc* algorithm to handle timing constraints.

	1C-TAs	2C-TAs	TAs
	real-time aut.		[ACD93,CY92]
Reachability	NLOGSPACE-C	NP-hard	PSPACE-C
$TCTL_{\leq,\geq}$ model checking	P-complete	PSPACE-C	PSPACE-C
TCTL model checking	PSPACE-complete		PSPACE-C

Fig. 5. Summary of the results

In timed model checking, an important challenge consists in developing data structures enabling to manage complexity blow-up due to timing constraints and to parallel composition of components; indeed it would be very interesting to have the benefits of DBMs for the timing constraints and those of BDDs for the control state explosion, but today no convincing solution exists. Our results motivate research for algorithms and data structures for simply timed systems composed by a unique clock and a parallel composition of processes. Of course, analysing such systems is PSPACE-hard due to the composition, nevertheless efficient data structures for handling such systems could be more easily defined due to the simple timing constraints.

References

- [ACD93] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.
- [ACH94] R. Alur, C. Courcoubetis, and T. A. Henzinger. The observational power of clocks. In Proc. 5th Int. Conf. Theory of Concurrency (CONCUR'94), Uppsala, Sweden, Aug. 1994, volume 836 of Lecture Notes in Computer Science, pages 162–177. Springer, 1994.
- [AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
- [AFH96] R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.
- [AH92] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In *Real-Time: Theory in Practice, Proc. REX Workshop, Mook, NL, June 1991,* volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer, 1992.
- [AH93] R. Alur and T. A. Henzinger. Real-time logics: Complexity and expressiveness. *Information and Computation*, 104(1):35–77, 1993.
- [AH94] R. Alur and T. A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–203, 1994.
- [AL02] L. Aceto and F. Laroussinie. Is your model checker on time? On the complexity of model checking for timed modal logics. *Journal of Logic and Algebraic Programming*, 52–53:7–51, 2002.
- [Alu91] R. Alur. Techniques for Automatic Verification of Real-Time Systems. PhD thesis, Stanford Univ., August 1991. Available as Tech. Report STAN-CS-91-1378.

- [Bou04] P. Bouyer. Forward analysis of updatable timed automata. Formal Methods in System Design, 24(3):281–320, 2004.
- [CC95] S. Campos and E. M. Clarke. Real-time symbolic model checking for discrete time models. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-Time System Development*, volume 2 of *AMAST Series in Computing*, pages 129–145. World Scientific, 1995.
- [CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
- [CY92] C. Courcoubetis and M. Yannakakis. Minimum and maximum delay problems in real-time systems. *Formal Methods in System Design*, 1(4):385–415, 1992.
- [Dil90] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In Proc. Int. Workshop Automatic Verification Methods for Finite State Systems (CAV'89), Grenoble, June 1989, volume 407 of Lecture Notes in Computer Science, pages 197–212. Springer, 1990.
- [Dim00] Catalin Dima. Real-time automata and the Kleene algebra of sets of real numbers. In Proc. of STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science, Lille, France, February 2000, volume 1770 of Lecture Notes in Computer Science, pages 279–289, 2000.
- [Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier Science, 1990.
- [GJ79] M. R. Garey and D. S. Johnson. Computers and Intractability. A Guide to the Theory of NP-Completeness. Freeman, 1979.
- [Hen98] T. A. Henzinger. It's about time: real-time logics reviewed. In Proc. 9th Int. Conf. Concurrency Theory (CONCUR'98), Nice, France, Sep. 1998, volume 1466 of Lecture Notes in Computer Science, pages 439–454. Springer, 1998.
- [HJ96] Dang Van Hung and Wang Ji. On the design of hybrid control systems using automata models. In Proc. 16th Conf. Found. of Software Technology and Theor. Comp. Sci. (FST&TCS'96), Hyderabad, India, Dec. 1996, volume 1180 of Lecture Notes in Computer Science, pages 156–167. Springer, 1996.
- [LMS02] F. Laroussinie, N. Markey, and Ph. Schnoebelen. On model checking durational Kripke structures (extended abstract). In Proc. 5th Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'2002), Grenoble, France, Apr. 2002, volume 2303 of Lecture Notes in Computer Science, pages 264–279. Springer, 2002.

A (End of) proof of Theorem 5.2

 $\xi = \mathbf{E}\varphi \mathbf{U}_{\leq c} \psi$: We use a kind of Bellman-Ford algorithm to compute the δ^{ψ} , it uses the natural min operation over duration functions: $\min((c_1, c_2), (c'_1, c'_2)) \stackrel{\text{def}}{=} (\min(c_1, c'_1), \min(c_2, c'_2))$. The duration functions $d^{\psi}_{q,\lambda}$ are first initialized to 0 if $\lambda \subseteq \operatorname{Sat}[q, \psi]$ or to ∞ otherwise (no path leading to ψ states has been yet discovered) and then we use the following procedure:

$$\begin{split} & \text{For } i = 1 \text{ to } |V_G| - 1 \text{ do} \\ & \text{For any } (q, \lambda) \to_G (q', \lambda') \text{ do} \\ & \text{ if } \left(q = q' \land \lambda' = \text{Succ}(\lambda) \land \lambda = (b_j; b_{j+1}) \land d_{q,\lambda'}^{\psi} = (c', c') \right) \\ & \text{ Then } // \text{ delay transition } - 1 \\ & d_{q,\lambda}^{\psi} = \min(d_{q,\lambda}^{\psi}, (b_{j+1} - b_j + c', c')) \\ & \text{ else if } \left(q = q' \land \lambda' = \text{Succ}(\lambda) \land \lambda = [b_j; b_j] \land d_{q,\lambda'}^{\psi} = (c', c'') \right) \\ & \text{ Then } // \text{ delay transition } - 2 \\ & d_{q,\lambda}^{\psi} = \min(d_{q,\lambda}^{\psi}, (c', c')) \\ & \text{ else } // \text{ action transition} \\ & \text{ if } (\lambda' = \lambda \lor \lambda' = [0; 0]) \\ & \text{ Then } d_{q,\lambda}^{\psi} = \min(d_{q,\lambda}^{\psi}, d_{q,\lambda'}^{\psi}) \end{split}$$

Then it remains to build $\operatorname{Sat}[q, \xi]$ from the duration functions δ_q^{ψ} and the threshold $c: x \in \operatorname{Sat}[q, \xi]$ iff $\delta_{q,\lambda}^{\psi}(x) \leq c$. This may lead to cut an interval in two parts. A crucial point of the algorithm is to merge as much as possible these (fragments of) *G* intervals, and we have to show that the size of $\operatorname{Sat}[q, \xi]$ can be bounded enough to ensure a polynomial algorithm. We are going to bound (1) the number of intervals of $\operatorname{Sat}[q, \xi]$ coming from a given interval of $\operatorname{Sat}[q, \varphi]$ and (2) the number of new constants (not present in **B**) that can appear due to the cuts.

Consider an interval I of $\operatorname{Sat}[q, \varphi]$. This interval corresponds to a finite sequence of G-nodes $(q, \lambda_1), \dots, (q, \lambda_k)$ s.t. $\lambda_{i+1} = \operatorname{Succ}(\lambda_i)$. The threshold " $\leq c$ " may cut these intervals and provide non-adjacent intervals in $\operatorname{Sat}[q, \xi]$. We can distinguish two cases of cuts: (1) the cut is done between two intervals, or (2) the cut is done inside an interval. In both cases the cut is due to a unique constraint in a transition $(x < \operatorname{or} x >)$ which can only cut this interval. Since a transition may contain at most two such constraints, the size of $\operatorname{Sat}[q, \xi]$ will be bounded by $|\operatorname{Sat}[q, \psi]| + (|\operatorname{Sat}[q, \varphi]| + 2 \cdot |\rightarrow_A|)$. Indeed:

- Consider a cut between two intervals λ_j and λ_{j+1}. Assume λ_j = [b_i; b_i] and λ_{j+1} = (b_i, b_{i+1}). Moreover assume δ^ψ_{q,λj} = (c₀, c₀) and δ^ψ_{q,λj} = (c₁, c₂). If there is a cut in b_i, then c₀ < c < c₁. The shortest paths enabled from b_i do not exist from (b_i, b_{i+1}) and then these SPs start by a sequence of action transitions and one of them (performed before any delay and reset) have a guard x ≤ b_i. This transition can only cut in b_i the intervals of Sat[q, φ]. The case λ_j = (b_i; b_{i+1}) and λ_{j+1} = [b_{i+1}, b_{i+1}] is similar.
- 2. Consider a cut inside an interval $\lambda_j = (b_i; b_{i+1})$. Then the cut occurs in the decreasing part of $\delta_{q,\lambda_j}^{\psi} = (c_1, c_2)$ and we have $c_1 > c > c_2$. The cut occurs in $b_{i+1} (c-c_2)$, and introduces a new (integer) constant. For the valuations in the decreasing part, *i.e.* between $b_{i+1} (c_1 c_2)$ and b_{i+1} , the shortest paths have delay transitions before any reset. This required delay is due to a guard of the form x > k or $x \ge k$ along the SP. Such a constraint induces the cut and only this one (in configurations of the form (\cdot, λ_i)).

Therefore a guard m < x < M may induce at most two cuts in $\mathsf{Sat}[q,\xi]$, then $|\mathsf{Sat}[q,\xi]| \leq |\mathsf{Sat}[q,\psi]| + |\mathsf{Sat}[q,\varphi]| + 2 \cdot |\to_A|$. And it creates at most one new integer constant (this also holds for the other modalities) and this entails $\mathbb{B} \leq \mathsf{Cst}_A + |\to_A| \cdot |\xi|$. Finally the complexity of the procedure is in $O(|V_G| \cdot |\to_G|)$, with $|V_G| \leq |Q_A| \cdot 2 \cdot |\mathbb{B}|$ and $|\to_G| \leq (|\to_A| + 1) \cdot |V_G|$. This provides an algorithm in $O(|Q_A|^2 \cdot |\to_A|^3 \cdot |\xi|^2)$ for $\mathsf{E}_{-}\mathsf{U}_{\leq c}$.

 $\xi = \mathsf{E}\varphi \mathsf{U}_{\geq c} \psi$: For building $\mathsf{Sat}[q, \mathsf{E}\varphi \mathsf{U}_{\geq c} \psi]$, we use the same idea as for $\mathsf{E}\varphi \mathsf{U}_{\leq c} \psi$ formula based on the graph $G = (V_G, \rightarrow_G, l_G)$ but here we label nodes by $\varphi \land \neg \psi, \varphi \land \psi$ and $\neg \varphi \land \psi$. We restrict ourself to nodes satisfying $\mathcal{E}\varphi \mathsf{U}\psi$ and we introduce a new atomic proposition $P_{\mathsf{scc}^+(\varphi)}$ in order to label every node (q, λ) in *G* belonging to a strongly connected set of nodes satisfying φ and where at least one edge is an abstract delay transition. Labeling states for $P_{\mathsf{scc}^+(\varphi)}$ can be done in time O(|G|) once they are labeled for φ .

We can now solve the original problem. There are two ways a state can satisfy ξ :

- Either a path with loops is required so that a long enough duration is reached: such a state verifies the *CTL* formula $E\varphi UP_{scc^+(\varphi)}$ since any *G* state satisfies $E\varphi U\psi$. This can be done in $O(|V_G| + |\rightarrow_G|)$.
- Or a simple path is enough. Then we can use a (simple) variant of the earlier shortest paths method, this times geared towards *longest acyclic paths* (LAP). For this we just remove states labeled by $P_{\text{SCC}^+}(\varphi)$, consider states satisfying $\neg \varphi \land \psi$ as final states and remove loops with null durations. The algorithm runs in $O(|V_G| + |\rightarrow_G|)$ and we keep (sub-)intervals whose LAP is above the threshold *c*.

Finally we build $\operatorname{Sat}[q,\xi]$ by merging the states of G satisfying $\operatorname{E}\varphi \bigcup_{\geq c} \psi$. As for labeling $\operatorname{E}\varphi \bigcup_{\leq c} \psi$, the procedure may add new constants (in the second case) and split intervals of $\operatorname{Sat}[q,\varphi]$ into several intervals in $\operatorname{Sat}[q,\xi]$ but we can argue as in the previous case and show that the size of $\operatorname{Sat}[q,\xi]$ is bounded by $|\operatorname{Sat}[q,\varphi]| + 2 \cdot |\to_A|$. The procedure runs in $O(|V_G| + |\to_G|)$, with $|V_G| \leq |Q_A| \cdot 2 \cdot |\mathbb{B}|$, $\mathbb{B} \leq \operatorname{Cst}_A + |\to_A| \cdot |\xi|$ and $|\to_G| \leq (|\to_A| + 1) \cdot |V_G|$. This gives an algorithm in $O(|Q_A| \cdot |\to_A|^2 \cdot |\xi|)$.

 $\xi = A\varphi U_{\leq c}\psi$: We reduce to the previous cases using the following equivalences

$$\begin{aligned} \mathsf{A}\varphi\mathsf{U}_{\leq c}\,\psi &\equiv \mathsf{AF}_{\leq c}\,\psi \wedge \neg\mathsf{E}(\neg\psi)\mathsf{U}(\neg\psi \wedge \neg\varphi)\\ \mathsf{AF}_{< c}\,\psi &\equiv \mathsf{AF}\psi \wedge \neg\mathsf{E}\neg\psi\mathsf{U}_{> c}\,\top. \end{aligned}$$

 $\xi = A\varphi U_{\geq c} \psi$: We use the equivalence $A\varphi U_{\geq c} \psi \equiv AG_{<c} (A\varphi U_{>0} \psi)$. And it is easy to write a labeling procedure for $A\varphi U_{>0} \psi$ over the same *G*-graph used for $E\varphi U_{\geq c} \psi$: A node verifies $A\varphi U_{>0} \psi$ iff it verifies $AG_{\leq 0} \varphi$, $A\varphi U\psi$ and after the first abstract delay transition $A\varphi U\psi$ has to hold.

Strict subscripts: The modalities with $\langle c | or \rangle c$ are treated as the previous ones.

On Flatness for 2-Dimensional Vector Addition Systems with States

Jérôme Leroux¹* and Grégoire Sutre²

 ¹ DIRO, Université de Montréal, Montréal, QC, Canada leroujer@iro.umontreal.ca
² LaBRI, Univ. de Bordeaux & CNRS UMR 5800, Talence, France sutre@labri.fr

Abstract. Vector addition systems with states (VASS) are counter automata where (1) counters hold nonnegative integer values, and (2) the allowed operations on counters are increment and decrement. Accelerated symbolic model checkers, like FAST, LASH or TReX, provide generic semi-algorithms to compute reachability sets for VASS (and for other models), but without any termination guarantee. Hopcroft and Pansiot proved that for 2-dim VASS (i.e. VASS with two counters), the reachability set is effectively semilinear. However, they use an ad-hoc algorithm that is specifically designed to analyze 2-dim VASS. In this paper, we show that 2-dim VASS are *flat* (i.e. they "intrinsically" contain no nested loops). We obtain that — forward, backward and binary — reachability sets are effectively semilinear for the class of 2-dim VASS, and that these sets can be computed using generic acceleration techniques.

1 Introduction

Distributed systems have regained much attention recently, especially due to the popularization of the Internet. Ensuring correctness of distributed systems is usually challenging, as these systems may contain subtle errors that are very hard to find. To overcome this difficulty, a *formal verification* approach can be employed: model the system, model the desired property, and algorithmically check that the system satisfies the property.

Petri nets, and equivalently *vector addition systems with states* (VASS), are a widely used formalism to model concurrent distributed systems. Basically, a VASS is a *counter automaton* where (1) counters hold nonnegative integer values, and (2) the allowed operations on counters are *increment* and *decrement*. As the counters are unbounded, VASS are naturally *infinite-state* systems.

Various formalisms have been proposed to model desired properties on systems. In this work, we only consider *safety* properties: these properties (of the original system) may often be expressed by *reachability properties* on the model.

© Springer-Verlag Berlin Heidelberg 2004

^{*} This work was partly carried out during the first author's doctoral studies at Lab. Specification and Verification, ENS de Cachan (France).

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 402-416, 2004.

Reachability properties are algorithmically checkable for *finite-state* systems (and efficient implementations exist). However, the situation is more complex for *infinite-state* systems: the reachability problem is undecidable even for restricted classes of systems, such as Minsky machines [Min67].

Accelerated Symbolic Model-Checking. Verification of reachability properties usually proceeds through an iterative fixpoint computation of the *forward reachability set* post*(resp. *backward reachability set* pre*), starting from the initial states (resp. from the error states). When the state space is infinite, finite *symbolic representations* for sets of states are required [HM00]. To help termination of this fixpoint computation, so-called *acceleration* techniques (or *meta-transitions*) are applied [BW94, BGWW97, BH99, BJNT00, FIS03, FL02]. Basically, acceleration consists in computing in one step the effect of iterating a given loop (of the control flow graph). Accelerated symbolic model checkers such as LASH [Las], TReX [ABS01], and FAST [BFLP03] implement this approach.

Accelerated symbolic model-checking is only a *semi*-algorithm: it does not provide any guarantee of termination. Still, this approach is very promising as it behaves well in practice. It would be desirable to determine some classes of systems for which termination is guaranteed. A natural sufficient condition for termination is *flatness* [CJ98]: a system S is called *flat*¹ when we can "extract" from S a finite number of subsystems S'_0, \ldots, S'_n such that (1) each S'_i contains no nested loops, and (2) reachability in S is equivalent to reachability in $\bigcup_i S'_i$. When the system is flat, and if every loop can be accelerated, then (a symbolic representation of) post* / pre* is computable. In fact, flatness is also a necessary condition for termination of acceleration-based semi-algorithms. Hence, flatness turns out to be a crucial notion for termination analysis of accelerated symbolic model-checking.

Dedicated Algorithms for VASS. Many specialized algorithms have been designed to solve verification problems for various extensions and restrictions of VASS. The reachability problem for VASS has been proved decidable [May84, Kos82]. The reachability sets **post*** and **pre*** are effectively semilinear for Lossy VASS [BM99]. The class of 2-dimensional VASS (i.e. VASS with only two counters) has received much attention. Hopcroft and Pansiot proved that the reachability sets **post*** and **pre*** are effectively semilinear for this class [HP79]. It was later shown that **post* / pre*** are still effectively semilinear for various extensions of 2-dim VASS [FS00b, FS00a]. However, these methods suffer from serious drawbacks: (1) they cannot be easily extended or combined, and (2) from an implementation perspective, a dedicated tool would be needed for each specialized algorithm.

Our Contribution. Recall that the reachability sets **post*** and **pre*** have been shown to be effectively semilinear for the class of 2-dim VASS [HP79]. In this

¹ Our notion of flatness is actually more general than in [CJ98]: there, a system is called flat when it contains no nested loops.

paper, we investigate termination of the generic acceleration-based computation of **post*/pre*** for this class. The reader familiar with Petri nets will observe that our results also hold for the class of Petri nets having only two unbounded places. Our main result is the following:

Every 2-dim VASS is flat

Since every loop in a VASS can be accelerated [CJ98, FL02], we also obtain the following new results:

- *i*) the binary reachability relation \mathcal{R}^* is effectively semilinear for the class of 2-dim VASS.
- *ii*) the semilinear sets **post**^{*}, **pre**^{*} and \mathcal{R}^* of any 2-dim VASS can be computed using generic acceleration techniques.

In particular, we get that accelerated symbolic model checkers such as LASH, TReX, or FAST, terminate on 2-dim VASS (if a suitable search strategy is used). From a practical viewpoint, our approach has several benefits: (1) we can apply a generic algorithm, which was designed for a much larger class of systems, with no need for a preliminary syntactic check, and (2) the sets **post***, **pre*** and \mathcal{R}^* can be computed using the same algorithm. For instance, we directly obtain that all six VASS examples given in [BLW03] are flat, and hence their binary reachability relation can be computed by means of acceleration.

The effective semilinearity of the binary reachability relation \mathcal{R}^* is also a surprising theoretical result, which can prove useful in pratice. Indeed, we may express many properties on the model as a first order formula over states using (binary) predicates \mathcal{R} and \mathcal{R}^* . For instance, we can check relationships between input values and output values for a 2-dim VASS modeling a function. We may also use this computation of \mathcal{R}^* for *parameter synthesis:* for instance, we can compute the set of initial states such that the counters stay bounded, or such that the system terminates. We may also express and check subtle properties on the system's strongly connected components (SCC).

Our results obviously extend to the class of (arbitrary) VASS where only two counters are modified. To guide the analysis of a complex modular VASS, we may choose to replace some flat subsystems (e.g. subsystems manipulating only two counters) by equivalent semilinear meta-transitions. Such a flatness-guided approach would surely help termination of accelerated symbolic model checkers.

Outline. The paper is organized as follows. Section 2 presents vector addition systems with states. We introduce the notion of flatness in Section 3 and we show that the binary reachability relation of any flat VASS is effectively semilinear. As a first step towards flatness, Section 4 investigates the notion of ultimate flatness. Finally, in Section 5, we focus on dimension 2 and we prove our main result: every 2-dim VASS is flat.

Some proofs had to be omitted due to space constraints. A self-contained long version of this paper (with detailed proofs for all results) can be obtained from the authors.

2 Vector Addition Systems with States (VASS)

This section is devoted to the presentation of vector addition systems with states. We first give basic definitions and notations that will be used throughout the paper.

2.1 Numbers, Vectors, Relations

Let \mathbb{Z} (resp. $\mathbb{N}, \mathbb{Z}^-, \mathbb{Q}, \mathbb{Q}_+$) denote the set of *integers* (resp. *nonnegative integers*, *nonnositive integers*, *rational numbers*, *nonnegative rational numbers*). We denote by \leq the usual total order on \mathbb{Q} . Given $k, l \in \mathbb{N}$, we write $[k \dots l]$ (resp. $[k \dots \infty[)$ for the *interval of integers* $\{i \in \mathbb{N} \mid k \leq i \leq l\}$ (resp. $\{i \in \mathbb{N} \mid k \leq i\}$). We write |X| the *cardinal* of any finite set X.

Given a set X and $n \in \mathbb{N}$, we write X^n for the set of *n*-dim vectors x of elements in X. For any index $i \in [1 ... n]$, we denote by x[i] the i^{th} component of an *n*-tuple x.

We now focus on *n*-dim vectors of (integer or rational) numbers. We write 0 for the *all zero vector*: 0[i] = 0 for all $i \in [1 .. n]$. We also denote by \leq the *usual partial order on* \mathbb{Q}^n , defined by $x \leq y$ if for all $i \in [1 .. n]$ we have $x[i] \leq y[i]$.

Operations on *n*-dim vectors are componentwise extensions of their scalar counterpart (e.g. for $x, x' \in \mathbb{Q}^n$, x + x' is the vector $y \in \mathbb{Q}^n$ defined by y[i] = x[i] + x'[i] for all $i \in [1..n]$). For $\alpha \in \mathbb{Q}$ and $x \in \mathbb{Q}^n$, αx is the vector $y \in \mathbb{Q}^n$ defined by $y[i] = \alpha x[i]$ for all $i \in [1..n]$.

These operations are classically extended on sets of *n*-dim vectors (e.g. for $P, P' \subseteq \mathbb{Q}^n$, $P + P' = \{p + p' | p \in P, p' \in P'\}$). Moreover, in an operation involving sets of *n*-dim vectors, we shortly write x for the singleton $\{x\}$ (e.g. for $P \subseteq \mathbb{Q}^n$ and $x \in \mathbb{Q}^n$, we write x + P for $\{x\} + P$).

A binary relation R on some set X is any subset of $X \times X$. We shortly write x R x' whenever $(x, x') \in R$. Given two binary relations R_1, R_2 on X, the composed binary relation $R_1 \cdot R_2$ on X is defined by $x (R_1 \cdot R_2) x'$ if we have $x R_1 y$ and $y R_2 x'$ forsome $y \in X$. We denote by R^* the reflexive and transitive closure of R. The identity relation on X is the binary relation $Id_X = \{(x, x) | x \in X\}$. In the rest of the paper, we will only consider binary relations, and they will shortly be called relations.

2.2 Vector Addition Systems with States

Definition 2.1. An *n*-dim vector addition system with states (VASS for short) is a 5-tuple $V = (Q, T, \alpha, \beta, \delta)$ where Q is a finite non empty set of locations, T is a finite non empty set of transitions, $\alpha : T \to Q$ and $\beta : T \to Q$ are the source and target mappings, and $\delta : T \to \mathbb{Z}^n$ is a transition displacement labeling.

An *n*-dim VASS is basically a finite graph whose edges are labeled by *n*-dim vectors of integers. Each component $i \in [1 .. n]$ corresponds to a counter ranging over N. Operationally, control flows from one location to another along transitions, and counters simultaneously change values by adding the transition's displacement (as long as the counters remain nonnegative).

Formally, let $V = (Q, T, \alpha, \beta, \delta)$ be an *n*-dim VASS. The set of configuration C_V of V is $Q \times \mathbb{N}^n$, and the semantics of each transition $t \in T$ is given by the transition reachability relation $\mathcal{R}_V(t)$ over \mathcal{C}_V defined by:

$$(q, \mathsf{x}) \mathcal{R}_V(t) (q', \mathsf{x}')$$
 if $q = \alpha(t), q' = \beta(t)$ and $\mathsf{x}' = \mathsf{x} + \delta(t)$

We write T^+ for the set of all *non empty words* $t_0 \cdots t_k$ with $t_i \in T$, and ε denotes the *empty word*. The set $T^+ \cup \{\varepsilon\}$ of all *words* π over T is denoted by T^* . Transition displacements and transition reachability relations are naturally extended to words:

$$\begin{cases} \delta(\varepsilon) = 0\\ \delta(\pi \cdot t) = \delta(\pi) + \delta(t) \end{cases} \qquad \qquad \begin{cases} \mathcal{R}_V(\varepsilon) = Id_{\mathcal{C}_V}\\ \mathcal{R}_V(\pi \cdot t) = \mathcal{R}_V(\pi) \cdot \mathcal{R}_V(t) \end{cases}$$

A *language* over *T* is any subset *L* of *T*^{*}. We also extend displacements and reachability relations to languages: $\delta(L) = \{\delta(\pi) \mid \pi \in L\}$ and $\mathcal{R}_V(L) = \bigcup_{\pi \in L} \mathcal{R}_V(\pi)$.

Definition 2.2. Given a VASS $V = (Q, T, \alpha, \beta, \delta)$, the one-step reachability relation of V is the relation $\mathcal{R}_V(T)$, shortly written \mathcal{R}_V . The global reachability relation of V is the relation $\mathcal{R}_V(T^*)$, shortly written \mathcal{R}_V^* .

Remark that the global reachability relation is the reflexive and transitive closure of the one-step reachability relation. The global reachability relation of a VASS V is also usually called *the binary reachability relation* of V. A *reachability subrelation* is any relation $R \subseteq \mathcal{R}_V^*$. For the reader familiar with transition systems, the operational semantics of V can be viewed as the infinite-state transition system (C_V, \mathcal{R}_V).

Consider two locations q and q' in a VASS V. A word $\pi \in T^*$ is called a *path* from q to q' if either (1) $\pi = \varepsilon$ and q = q', or (2) $\pi = t_0 \cdots t_k$ and satisfies: $q = \alpha(t_0), q' = \beta(t_k)$ and $\beta(t_{i-1}) = \alpha(t_i)$ for every $i \in [1 \dots k]$. A path from q to q is called a *loop on* q, or shortly a *loop*. We denote by $\Pi_V(q,q')$ the set of all paths from q to q' in V. The set $\bigcup_{q,q' \in Q} \Pi_V(q,q')$ of all *paths* in V is written Π_V .

Notation. In the following, we will simply write \mathcal{R} (resp. Π , \mathcal{C}) instead of \mathcal{R}_V (resp. Π_V, \mathcal{C}_V), when the underlying VASS is unambiguous. We will also sometimes write \rightarrow (resp. $\xrightarrow{\pi}$, \xrightarrow{L} , $\xrightarrow{*}$) instead of \mathcal{R} (resp. $\mathcal{R}(\pi), \mathcal{R}(L), \mathcal{R}^*$).

We will later use the following fact, which we leave unproved as it is a well known property of VASS. Recall that a *prefix* of a given word $\pi \in T^*$ is any word σ such that $\pi = \sigma \cdot \sigma'$ for some word σ' .

Fact 1. For any configurations (q, x) and (q', x') of a VASS V, and for any word $\pi \in T^*$, we have:

$$(q, \mathsf{x}) \xrightarrow{\pi} (q', \mathsf{x}') \quad iff \quad \begin{cases} \pi \in \Pi(q, q') \\ \mathsf{x}' = \mathsf{x} + \delta(\pi) \\ \mathsf{x} + \delta(\sigma) \ge 0 \text{ for every prefix } \sigma \text{ of } \pi \end{cases}$$

TEAM LING



Fig. 1. A 3-dim VASS weakling computing the powers of 2

Observe that for any word $\pi \in T^*$, the relation $\mathcal{R}_V(\pi)$ is non empty iff π is a path.

Example 2.3. Consider the 3-dim VASS *E* depicted on Figure 1. This example is a variation of an example in [HP79]. Formally, this VASS is the 5-tuple $E = (\{p,q\}, \{t_1, t'_1, t_2, t_3, t_4\}, \alpha, \beta, \delta)$ where $\alpha(t_1) = \alpha(t'_1) = \alpha(t_2) = \beta(t_1) = \beta(t'_1) = \beta(t_4) = p$, and $\alpha(t_3) = \alpha(t_4) = \beta(t_3) = \beta(t_2) = q$, and δ is defined by: $\delta(t_1) = -\delta(t'_1) = (0, 1, -1), \ \delta(t_2) = (0, 0, 0), \ \delta(t_3) = (0, -1, 2)$ and $\delta(t_4) = (-1, 0, 0)$.

Intuitively, the loop t_1 on p transfers the contents of the third counter into the second counter, while the loop t_3 on q transfers twice as much as the contents of the second counter into the third counter. However, the VASS may change location (using transition t_2 or t_4) before the transfer completes (a "zero-test" would be required to ensure that the transfer always completes). Transition t_2 acts as a "silent transition", and transition t_4 decrements the first counter by 1. The loop t'_1 on p has been added to simplify the expression of \mathcal{R}^* .

Consider the path $\pi = t_1 t_1 t_2 t_3 t_3 t_4 t_2$. It is readily seen that the reachability subrelation $\mathcal{R}(\pi)$ is precisely the set of pairs ((p, (x+1, y, z+2)), (q, (x, y, z+4))) with $(x, y, z) \in \mathbb{N}^3$. This little VASS exhibits a rather complex global reachability relation, since it can be proved² that: $(p, (x, y, z)) \xrightarrow{*} (p, (x', y', z'))$ iff $x' \leq x$, $y + z \leq y' + z'$, and $2^{x'}(y' + z') \leq 2^{x}(y + z)$.

3 Effective Semilinearity of \mathcal{R}^* for Flat VASS

An important concept used in this paper is that of *semilinear sets* [GS66]. For any subset $P \subseteq \mathbb{Z}^n$, we denote by P^* the set of all (finite) linear combinations of vectors in P:

$$P^* = \left\{ \sum_{i=0}^k c_i \, \mathsf{p}_i \ / \ k, c_0, \dots, c_k \in \mathbb{N} \text{ and } \mathsf{p}_0, \dots, \mathsf{p}_k \in P \right\}$$

A subset $S \subseteq \mathbb{Z}^n$ is said to be a *linear set* if $S = (x + P^*)$ for some $x \in \mathbb{Z}^n$ and for some finite subset $P \subseteq \mathbb{Z}^n$; moreover x is called the *basis* and vectors in P are called *periods*. A *semilinear set* is any finite union of linear sets. Let us

² This proof is an adpatation of the proof in [HP79], and is left to the reader.

recall that semilinear sets are precisely the subsets of \mathbb{Z}^n that are definable in Presburger arithmetic $\langle \mathbb{Z}, \leq, + \rangle$ [GS66].

Observe that any finite non empty set Q can be "encoded" using a bijection η from Q to [1..|Q|]. Thus, these semilinearity notions naturally carry³ over subsets of $Q \times \mathbb{Z}^n$ and over relations on $Q \times \mathbb{Z}^n$.

Definition 3.1. A linear path scheme (LPS for short) for a VASS V is any language $\rho \subseteq \Pi_V$ of the form $\rho = \sigma_0 \theta_1^* \sigma_1 \cdots \theta_k^* \sigma_k$ where $\sigma_0, \theta_1, \sigma_1, \ldots, \theta_k, \sigma_k$ are words. A semilinear path scheme (SLPS for short) is any finite union of LPS.

Remark that a language of the form $\sigma_0 \theta_1^* \sigma_1 \cdots \theta_k^* \sigma_k$, with $\sigma_i, \theta_i \in T^*$, is an LPS iff (1) $\sigma_0 \theta_1 \sigma_1 \cdots \theta_k \sigma_k$ is a path, and (2) θ_i is a loop for every $i \in [1..k]$.

Definition 3.2. Given a VASS V, a reachability subrelation $R \subseteq \mathcal{R}_V^*$ is called flat if $R \subseteq \mathcal{R}_V(\rho)$ for some SLPS ρ . We say that V is flat when \mathcal{R}_V^* is flat.

The class of flat reachability subrelations is obviously closed under union and under composition.

From a computability viewpoint, any (finitely "encoded") set *S* is said to be *effectively semilinear* if (1) *S* is semilinear, and (2) a finite basis-period description (or equivalently a Presburger formula) for *S* can be computed (from its "encoding"). The following *acceleration theorem* shows that the reachability subrelation "along" any SLPS is an effectively semilinear set. This theorem was proved in [CJ98, FL02] for considerably richer classes of counter automata. We give a simple proof for the simpler case of VASS.

Theorem 3.3 ([CJ98, FL02]). For any SLPS ρ in a VASS V, the reachability subrelation $\mathcal{R}_V(\rho)$ is effectively semilinear.

Proof. Let V denote an *n*-dim VASS. Observe that for any transition t in V, the reachability subrelation $\mathcal{R}(t)$ is effectively semilinear. As the class of effectively semilinear reachability subrelations is closed under union and under composition, it suffices to show that $\mathcal{R}(\theta^*)$ is effectively semilinear for any loop θ . Consider a loop θ on some location q. It is readily seen that:

$$\begin{array}{lll} \mathcal{R}(\theta^*) &= & Id_{\{q\} \times \mathbb{N}^n} \\ & \bigcup \ \mathcal{R}(\theta) \\ & \bigcup \ \mathcal{R}(\theta) \cdot \{((q, \mathsf{x}), (q, \mathsf{x}')) \ / \ \mathsf{x}, \mathsf{x}' \in \mathbb{N}^n \ \text{and} \ \mathsf{x}' \in (\mathsf{x} + \{\delta(\theta)\}^*)\} \cdot \mathcal{R}(\theta) \end{array}$$

Hence we get that $\mathcal{R}(\theta^*)$ is effectively semilinear, which concludes the proof.

Corollary 3.4. The global reachability relation \mathcal{R}_V^* of any flat VASS V is effectively semilinear.

³ Obviously, the extension of these notions does not depend on the "encoding" η .

Proof. Assume that *V* is a flat VASS. Since *V* is flat, there exists an SLPS ρ satisfying $\mathcal{R}_V^* = \mathcal{R}_V(\rho)$. In order to compute such an SLPS, we may enumerate all SLPS ρ , and stop as soon as ρ satifies $Id_{\mathcal{C}_V} \cup \mathcal{R}_V(\rho) \cdot \mathcal{R}_V \subseteq \mathcal{R}_V(\rho)$. All required computations are effective: \mathcal{R}_V is readily seen to be effectively semilinear, semilinear relations are effectively closed by composition, and equality is decidable between semilinear relations. We then apply Theorem 3.3 on ρ . \Box

Moreover, the semilinear global reachability relation \mathcal{R}_V^* of any flat VASS *V* can be computed using an existing "accelerated" symbolic model checker such as LASH [Las], TREX [ABS01], or FAST [BFLP03]. In this paper, we prove that every 2-dim VASS is flat, and thus we get that the global reachability relation of any 2-dim VASS is effectively semilinear. This result cannot be extended to dimension 3 as the 3-dim VASS *E* of Example 2.3 has a non semilinear global reachability relation.

Given an *n*-dim VASS *V* and a subset $S \subseteq C_V$ of configurations, we denote by $\text{post}_V^*(S)$ the set $\{s' \in C_V \mid \exists s \in S, s \mathcal{R}_V^* s'\}$ of *successors* of *S*, and we denote by $\text{pre}_V^*(S)$ the set $\{s \in C_V \mid \exists s' \in S, s \mathcal{R}_V^* s'\}$ of *predecessors* of *S*. It is well known that for any 2-dim VASS *V*, the sets $\text{post}^*(S)$ and $\text{pre}^*(S)$ are effectively semilinear for every semilinear subset *S* of configurations [HP79]. One may be tempted to think that the semilinearity of \mathcal{R}_V^* is a consequence of this result. The following proposition shows that this is not the case.

Proposition 3.5. There exists a 3-dim VASS V such that (1) $\text{post}^*_V(S)$ and $\text{pre}^*_V(S)$ are effectively semilinear for every semilinear subset $S \subseteq C_V$, and (2) the global reachability relation \mathcal{R}^*_V is not semilinear.

4 Acceleration Works Better in Absence of Zigzags

The rest of the paper is devoted to the proof that every 2-dim VASS is flat. We first establish in this section some preliminary results that hold in any dimension. We will restrict our attention to dimension 2 in the next section.

It is well known that the set $\delta(\Pi(q, q'))$ of displacements of all paths between any two locations q and q' is a semilinear set. We now give a stronger version of this result: this set of displacements can actually be "captured" by an SLPS.

Lemma 4.1. For every pair (q, q') of locations in a VASS V, there exists an SLPS $\rho_{q,q'} \subseteq \Pi(q,q')$ such that $\delta(\rho_{q,q'}) = \delta(\Pi(q,q'))$.

Given any two locations q and q' in a VASS V, the "counter reachability subrelation" $\{(\mathbf{x}, \mathbf{x}') \in (\mathbb{N}^n)^2 / (q, \mathbf{x}) \xrightarrow{*} (q', \mathbf{x}')\}$ between q and q' is clearly contained in the relation $\{(\mathbf{x}, \mathbf{x}') \in (\mathbb{N}^n)^2 / \mathbf{x}' - \mathbf{x} \in \delta(\Pi(q, q'))\}$. According to the lemma, there exists an SLPS $\rho \subseteq \Pi(q, q')$ such that $\delta(\Pi(q, q')) = \delta(\rho)$. Still, $\mathcal{R}(\rho)$ does not necessarily contain the reachability subrelation between q and q', as shown by the following example.

Example 4.2. Consider again the VASS *E* of Example 2.3. The set of displacements $\delta(\Pi(p, p))$ is equal to $\delta(\rho)$ where ρ is the SLPS contained in $\Pi(p, p)$

defined by: $\rho = (t_1)^* (t'_1)^* \cup (t_1)^* (t'_1)^* t_2(t_3)^* t_4(t_2t_4)^*$. Note that $\delta(\rho)$ is the semilinear set $\delta(\rho) = P_1^* \cup ((-1,0,0) + P_2^*)$ with $P_1 = \{(0,1,-1), (0,-1,1)\}$ and $P_2 = P_1 \cup \{(0,-1,2), (-1,0,0)\}.$

It is readily seen that $\mathcal{R}(\rho)$ satisfies: $(p, (x, y, z)) \xrightarrow{\rho} (p, (x', y', z'))$ iff either (1) x' = x and y' + z' = y + z, or (2) x' < x and $y + z \le y' + z' \le 2(y + z) - y'$. Hence, according to Example 2.3, $\mathcal{R}(\rho)$ does not contain all pairs ((p, x), (p, x')) such that $(p, x) \xrightarrow{*} (p, x')$.

As a first step towards flatness, we now focus on reachability between configurations that have "big counter values". This leads us to the notion of *ultimate flatness*, but we first need some new notations.

Notation. Consider an *n*-dim VASS V with a set of locations Q, and let R denote any (binary) relation on $Q \times \mathbb{N}^n$. For any subset $X \subseteq \mathbb{N}^n$, the restriction of R to X, written $R|_X$, is the relation $R|_X = R \cap (Q \times X)^2$.

Definition 4.3. An *n*-dim VASS V is called ultimately flat if the restriction $\mathcal{R}_V^*|_{[c..\infty]^n}$ is flat for some $c \in \mathbb{N}$.

Remark 4.4. For any ultimately flat VASS *V*, there exists $c \in \mathbb{N}$ such that the restriction $\mathcal{R}_{V}^{*}|_{[c..\infty[^{n}]}$ is semilinear.

In the rest of this section, we give a sufficient condition for ultimate flatness. This will allow us to prove, in the next section, ultimate flatness of every 2-dim VASS. This sufficient condition basically consists in assuming a stronger version of Lemma 4.1 where the considered SLPS $\rho_{q,q'}$ are zigzag-free. In the following, we consider a fixed *n*-dim VASS $V = (Q, T, \alpha, \beta, \delta)$.

Definition 4.5. An LPS $\rho = \sigma_0 \theta_1^* \sigma_1 \cdots \theta_k^* \sigma_k$ is said to be zigzag-free if for every $i \in [1 \dots n]$, the k integers $\delta(\theta_1)[i], \dots, \delta(\theta_k)[i]$ have the same sign. A zigzag-free SLPS is any finite union of zigzag-free LPS.

Intuitively, an LPS ρ is zigzag-free iff the displacements of all loops in ρ "point" in the same hyperquadrant, where by hyperquadrant, we mean a subset of \mathbb{Z}^n of the form $Z_1 \times \cdots \times Z_n$ with $Z_i \in \{\mathbb{N}, \mathbb{Z}^-\}$.

The following lemma shows that the intermediate displacements along any path in a zigzag-free LPS ρ belong to fixed hypercube (that only depends on ρ and π). This result is not very surprising: since all loops in ρ "point" in the same "direction", the intermediate displacements along any path in ρ can not deviate much from this direction.

Lemma 4.6. Given any zigzag-free LPS ρ , there exists an integer $c \ge 0$ such that for every path $\pi \in \rho$, the displacement $\delta(\sigma)$ of any prefix σ of π satisfies: $\delta(\sigma)[i] \ge Min\{0, \delta(\pi)[i]\} - c$ for every $i \in [1..n]$.

We may now express, in Proposition 4.8, our sufficient condition for ultimate flatness. The proof is based on the following lemma.

Lemma 4.7. Let q, q' denote two locations, and let $\rho \subseteq \Pi(q, q')$ be any zigzagfree SLPS such that $\delta(\rho) = \delta(\Pi(q, q'))$. There exists $c \in \mathbb{N}$ such that for every $x, x' \in [c \dots \infty[^n, if (q, x) \xrightarrow{*} (q', x') then (q, x) \xrightarrow{\rho} (q', x').$

Proposition 4.8. Let V be a VASS. Assume that for every pair (q,q') of locations, there exists a zigzag-free SLPS $\rho_{q,q'} \subseteq \Pi(q,q')$ such that $\delta(\rho_{q,q'}) = \delta(\Pi(q,q'))$. Then V is ultimately flat.

Observe that all proofs in this section are constructive. From Lemma 4.1, we can compute for each pair (q,q') of locations an SLPS $\rho_{q,q'} \subseteq \Pi_V(q,q')$ such that $\delta(\rho_{q,q'}) = \delta(\Pi_V(q,q'))$. Assume that these SLPS can be effectively "straightened" into zigzag-free SLPS $\rho'_{q,q'} \subseteq \Pi_V(q,q')$ with the same displacements: $\delta(\rho'_{q,q'}) = \delta(\Pi_V(q,q'))$. Then we can compute an integer $c \in \mathbb{N}$ such that $\mathcal{R}_V^*|_{[c...\infty]^n}$ is contained in $\mathcal{R}_V(\rho)$, where $\rho = \bigcup_{q,q'} \rho_{q,q'}$. Consequently, we can conclude using the acceleration theorem, Theorem 3.3, that $\mathcal{R}_V^*|_{[c...\infty]^n}$ is effectively semilinear. We will prove in the next section that this "straightening" assumption holds in dimension 2.

5 Flatness of 2-Dim VASS

We now have all the necessary background to prove our main result. We first show that every 2-dim VASS is ultimately flat. We then prove that every 1-dim VASS is flat, and we finally prove that every 2-dim VASS is flat.

5.1 Ultimate Flatness in Dimension 2

In order to prove ultimate flatness of all 2-dim VASS, we will need the following technical proposition.

Proposition 5.1. For any finite subset P of \mathbb{Z}^2 , and for any vector $x \in P$, there exists two finite subsets B', P' of $(x + P^*) \cap \mathbb{N}^2$ such that:

$$(\mathsf{x} + P^*) \cap \mathbb{N}^2 = B' + (P' \cup (P \cap \mathbb{N}^2))^*$$

Of course, this proposition also holds in dimension 1. The following remark shows that the proposition does not hold in dimension 3 (nor in any dimension above 3).

Remark 5.2. Consider the linear set $(\mathbf{x} + P^*)$ with basis $\mathbf{x} = (1,0,0)$ and set of periods $P = \{(1,0,0), (0,1,-1), (0,-1,2)\}$. Observe that $(\mathbf{x} + P^*) \cap \mathbb{N}^3 =$ $(1,0,0) + \mathbb{N}^3$. Let B' and P' denote two finite subsets of $(1,0,0) + \mathbb{N}^3$. There exists $c \in \mathbb{N}$ such that $B' \subseteq [0..c]^3$, and hence $(1,c+1,0) \notin B' + (P' \cup \{(1,0,0)\})^*$. Therefore, there does not exist two finite subsets B', P' of $(\mathbf{x} + P^*) \cap \mathbb{N}^3$ such that $(\mathbf{x} + P^*) \cap \mathbb{N}^3 = B' + (P' \cup (P \cap \mathbb{N}^3))^*$.

TEAM LING

We may now prove that every 2-dim VASS is ultimately flat. We first show that any LPS in a 2-dim VASS can be "straightened" into a zigzag-free SLPS with the same displacements.

Lemma 5.3. For any location q of a 2-dim VASS V and for any LPS $\rho \subseteq \Pi(q,q)$, there exists a zigzag-free SLPS $\rho' \subseteq \Pi(q,q)$ such that $\delta(\rho) \subseteq \delta(\rho')$.

Proposition 5.4. Every 2-dim VASS is ultimately flat.

Remark 5.5. There exists a 3-dim VASS that is not ultimately flat. To prove this claim, consider VASS *E* from Example 2.3. For every $c \in \mathbb{N}$, the restriction $\mathcal{R}^*|_{[c..\infty[^n]}$ is clearly non semilinear. According to Remark 4.4, we conclude that *E* is not ultimately flat.

5.2 Flatness and Effective Semilinearity of \mathcal{R}^* for 1-Dim VASS

Let $V = (Q, T, \alpha, \beta, \delta)$ be any 1-dim VASS and let us prove that V is flat. Proposition 5.4 is trivially extended to 1-dim VASS as any 1-dim VASS is "equal" to a 2-dim VASS whose second counter remains unchanged. Therefore, V is ultimately flat, and hence there exists $c \in \mathbb{N}$ such that $\mathcal{R}^*|_{[c..\infty[}$ is flat. Let $c' = c + \operatorname{Max}_t(|\delta(t)|)$ and let us denote by F and F_{∞} the intervals F = [0..c'] and $F_{\infty} = [c..\infty[$.

Recall that $\mathcal{R}^*|_{F_{\infty}}$ is flat. The restriction $\mathcal{R}^*|_F$ is also flat since it is a finite reachability subrelation. As the class of flat reachability subrelations is closed under union and under composition, we just have to prove the following inclusion:

$$\mathcal{R}^* \subseteq (\mathcal{R}^*|_F \cup \mathcal{R}^*|_{F_\infty})^2$$

Assume that $(q, x) \xrightarrow{\pi} (q', x')$ for some path π . If $x, x' \leq c'$ or $x, x' \geq c$ then $((q, x), (q', x')) \in \mathcal{R}^*|_F$ or $((q, x), (q', x')) \in \mathcal{R}^*|_{F_{\infty}}$, which concludes the proof since $(\mathcal{R}^*|_F \cup \mathcal{R}^*|_{F_{\infty}})$ contains $Id_{Q \times N}$.

Now suppose that either (1) $x \leq c$ and x' > c', or (2) x > c' and $x' \leq c$, and consider the case (1) $x \leq c$ and x' > c'. Let σ be the longest prefix of π such that $x + \delta(\sigma) \leq c$. As $x + \delta(\pi) = x' > c' \geq c$, the prefix σ can not be equal to π . So the path π can be decomposed into $\pi = \sigma t \sigma'$ with $\sigma, \sigma' \in T^*$ and $t \in T$, and such that $x + \delta(\sigma) \leq c$ and $x + \delta(\sigma t) > c$. We have $(q, x) \xrightarrow{\sigma t} (q'', x'') \xrightarrow{\sigma'} (q', x')$ where $q'' = \beta(t)$ and $x'' = x + \delta(\sigma t)$. Remark that $x'' = x + \delta(\sigma) + \delta(t)$ and hence $x'' \leq c + \delta(t) \leq c'$. From $x, x'' \leq c'$, we deduce that $((q, x), (q'', x'')) \in \mathcal{R}^*|_F$, and as $x'', x' \geq c$, we obtain that $((q'', x''), (q', x')) \in \mathcal{R}^*|_{F_{\infty}}$. So far, we have proved that $((q, x), (q', x')) \in (\mathcal{R}^*|_F \cup \mathcal{R}^*|_{F_{\infty}})^2$.

This concludes the proof that V is flat. We have just proved the following theorem.

Theorem 5.6. Every 1-dim VASS is flat.

5.3 Flatness and Effective Semilinearity of \mathcal{R}^* for 2-Dim VASS

Let $V = (Q, T, \alpha, \beta, \delta)$ be any 2-dim VASS and let us prove that V is flat. According to Proposition 5.4, V is ultimately flat, and hence there exists $c \in \mathbb{N}$ such that $\mathcal{R}^*|_{[c..\infty[^2]}$ is flat. Let $c' = c + \operatorname{Max}_t\{|\delta(t)[1]|, |\delta(t)[2]|\}$ and let us denote by F and F_{∞} the intervals F = [0..c'] and $F_{\infty} = [c..\infty[$. The set \mathbb{N}^2 is covered by 4 subsets:

$$\mathbb{N}^2 = (F \times F) \cup (F_{\infty} \times F) \cup (F \times F_{\infty}) \cup (F_{\infty} \times F_{\infty})$$

Recall that $\mathcal{R}^*|_{F_{\infty} \times F_{\infty}}$ is flat. The restriction $\mathcal{R}^*|_{F \times F}$ is also flat since it is a finite reachability subrelation.

Lemma 5.7. The reachability subrelations $(\mathcal{R}|_{F \times N})^*$ and $(\mathcal{R}|_{N \times F})^*$ are flat.

Proof. We only prove that $(\mathcal{R}|_{F\times\mathbb{N}})^*$ is flat (the proof that $(\mathcal{R}|_{\mathbb{N}\times F})^*$ is flat is symmetric). Observe that this reachability subrelation is the reachability relation of a 2-dim VASS whose first counter remains in the finite set *F*. So the relation $(\mathcal{R}_V|_{F\times\mathbb{N}})^*$ is first shown to be "equal" to the reachability relation of the 1-dim VASS $V' = (Q', T', \alpha', \beta', \delta')$ defined as follows:

$$\begin{cases} Q' = Q \times F \\ T' = \{(t, f) \in T \times F / f + \delta(t)[1] \in F \} \end{cases} \qquad \begin{cases} \alpha'((t, f)) = (\alpha(t), f) \\ \beta'((t, f)) = (\beta(t), f + \delta(t)[1]) \\ \delta'((t, f)) = \delta(t)[2] \end{cases}$$

Observe that reachability in V and V' are closely related: for every $t \in T$, $q, q' \in Q$, and $(f, y), (f', y') \in F \times \mathbb{N}$, we have:

 $(q, (f, y)) \mathcal{R}_{V}(t) (q', (f', y'))$ iff $((q, f), y) \mathcal{R}_{V'}((t, f)) ((q', f'), y')$

Let $\varphi : T'^* \to T^*$ denote the letter morphism defined by $\varphi((t, f)) = t$. We deduce from the previous equivalence, that the two following assertions hold for every $\pi' \in T'^*$, $q, q' \in Q$, and $x, y, x', y' \in \mathbb{N}$:

$$\begin{array}{ll} (q,(x,y)) \ \mathcal{R}_{V}(\varphi(\pi)) \ (q',(x',y')) & \text{if} & ((q,x),y) \ \mathcal{R}_{V'}(\pi') \ ((q',x'),y') \\ (q,(x,y)) \ (\mathcal{R}_{V}|_{F \times \mathbf{N}})^{*} \ (q',(x',y')) & \text{iff} & ((q,x),y) \ \mathcal{R}_{V'}^{*} \ ((q',x'),y') \end{array}$$

As V' is a 1-dim VASS, Theorem 5.6 shows that there exists a SLPS ρ' for V' such that $\mathcal{R}_{V'}^* \subseteq \mathcal{R}_{V'}(\rho')$. The language $\rho = \varphi(\rho')$ is an SLPS for V. Let us prove that $(\mathcal{R}_V|_{F\times\mathbb{N}})^* \subseteq \mathcal{R}_V(\rho)$.

Consider $((q, (x, y)), (q', (x', y'))) \in (\mathcal{R}_V|_{F \times N})^*$. Since $(\mathcal{R}_V|_{F \times N})^*$ is "equal" to $\mathcal{R}_{V'}^*$, we obtain that $((q, x), y) \mathcal{R}_{V'}^*$ ((q', x'), y'). As $\mathcal{R}_{V'}^* \subseteq \mathcal{R}_{V'}(\rho')$, we get that there exists a path $(t_0, f_0) \cdots (t_k, f_k) \in \rho'$ such that the pair (((q, x), y),((q', x'), y')) belongs $\mathcal{R}_{V'}((t_0, f_0) \cdots (t_k, f_k))$. Recall that $\mathcal{R}_V(\varphi(\pi))$ "contains" $\mathcal{R}_{V'}((t_0, f_0) \cdots (t_k, f_k))$. We deduce that $(q, (x, y)) \mathcal{R}_V(t_0 \cdots t_k) (q', (x', y'))$. We have shown that $(\mathcal{R}_V|_{F \times N})^* \subseteq \mathcal{R}_V(\rho)$, which concludes the proof. \Box Let us denote by R_1 the reachability subrelation $R_1 = Id \cup \mathcal{R} \cup (\mathcal{R}|_{N \times F})^* \cup (\mathcal{R}|_{F \times N})^*$, where *Id* denotes the identity relation on $Q \times N^2$. Recall that we want to prove that *V* is flat. As the class of flat reachability subrelations is closed under union and under composition, we just have to prove the following "flatness witness" inclusion:

$$\mathcal{R}^* \subseteq R_1 \cdot (Id \cup \mathcal{R}^*|_{F_{\infty} \times F_{\infty}}) \cdot R_1 \cdot (Id \cup \mathcal{R}^*|_{F \times F}) \cdot R_1 \cdot (Id \cup \mathcal{R}^*|_{F_{\infty} \times F_{\infty}}) \cdot R_1$$

Consider two configurations (q, x) and (q', x'), and a path $\pi \in \Pi(q, q')$, such that $(q, x) \xrightarrow{\pi} (q', x')$. An *intermediate vector* for the triple $((q, x), \pi, (q', x'))$ is a vector x'' such that $x'' = x + \delta(\sigma)$ for some prefix σ of π with $\sigma \notin \{\varepsilon, \pi\}$. Observe that for any such intermediate vector x'', there exists a state $q'' \in Q$ and a decomposition of π into $\pi = \sigma\sigma'$ with $\sigma, \sigma' \neq \varepsilon$, satisfying:

$$(q,\mathsf{x}) \xrightarrow{\sigma} (q'',\mathsf{x}'') \xrightarrow{\sigma'} (q',\mathsf{x}')$$

Let $G = (F \times F) \cup (F_{\infty} \times F_{\infty})$. We first prove the following lemma.

Lemma 5.8. For any $(q, x) \xrightarrow{\pi} (q', x')$ such that there is no intermediate vector in G, we have $((q, x), (q', x')) \in R_1$.

Proof. Assume that $(q, x) \xrightarrow{\pi} (q', x')$ is such that there is no intermediate vector in *G*. Remark that we can assume that $\pi \notin T \cup \{\varepsilon\}$. The intermediate vectors x'' are either in $[c' + 1 ... \infty[\times [0 ... c - 1] \text{ or in } [0 ... c - 1] \times [c' + 1 ... \infty[$. Assume by contradiction that there exists both an intermediate vector in $[c'+1 ... \infty[\times [0 ... c - 1] \text{ and in } [0 ... c - 1] \times [c' + 1 ... \infty[$. So there exists $t \in T$ such that either $(q_1, x_1) \xrightarrow{t} (q_2, x_2)$ or $(q_2, x_2) \xrightarrow{t} (q_1, x_1)$ with $x_1 \in [c'+1 ... \infty[\times [0 ... c - 1] \text{ and } x_2 \in [0 ... c - 1] \times [c' + 1 ... \infty[$. Let us consider the case $(q_1, x_1) \xrightarrow{t} (q_2, x_2)$. We have $x_2 = x_1 + \delta(t)$. From $x_1[1] \ge c' + 1$, we obtain $x_2[1] \ge c' + 1 - (c' - c) \ge c + 1$ which contradicts $x_1[1] \le c - 1$. As the case $(q_2, x_2) \xrightarrow{t} (q_1, x_1)$ is symmetric, we have proved that we cannot have both an intermediate state in $[c' + 1 ... \infty[\times [0 ... c - 1]]$ and in $[0 ... c - 1] \times [c' + 1 ... \infty[$. By symmetry, we can assume that all the intermediate states are in $[c' + 1 ... \infty[\times [0 ... c - 1]]$. Let *t* be the first transition of π . As $x + \delta(t)$ is an intermediate state, we have $x + \delta(t) \in [c' + 1 ... \infty[\times [0 ... c - 1]]$. In particular, $x \in \mathbb{N} \times [0 ... c']$. Symmetrically, by considering the last transition of π , we deduce $x' \in \mathbb{N} \times [0 ... c']$. Therefore, we have proved that $((q, x), (q', x')) \in R_1$. □

We may now prove the "flatness witness" inclusion given above. Consider any two configurations (q, x) and (q', x') such that $(q, x) \xrightarrow{*} (q', x')$. There exists a path $\pi \in \Pi(q, q')$ such that $(q, x) \xrightarrow{\pi} (q', x')$. We are going to prove that there exists a prefix σ of π and a suffix σ' of π such that there is no intermediate vectors of $((q, x), \sigma, (q_1, x_1))$ or $((q_2, x_2), \sigma', (q', x'))$ in $F \times F$ and such that $((q_1, x_1), (q_2, x_1)) \in Id \cup \mathcal{R}^*|_{F \times F}$. If there is no intermediate vector of $(q, x) \xrightarrow{\pi} (q', x')$ in $F \times F$, then we can choose $\sigma = \pi$ and $\sigma' = \varepsilon$. So we can assume that there is at least one intermediate state in $F \times F$. Let σ be the least prefix of π such that there is no intermediate vector of $((q, x), \sigma, (q_1, x_1))$ in $F \times F$ and $x_1 \in F$, and let σ' be the least suffix of π such that there is no intermediate vector of $((q_2, x_2), \sigma', (q', x'))$ in $F \times F$ and $x_2 \in F$. Now, just remark that $((q_1, x_1), (q_2, x_2)) \in \mathcal{R}^*|_{F \times F}$.

By decomposing in the same way the two paths σ and σ' such that there is no intermediate vector in $F_{\infty} \times F_{\infty}$, we have proved that for any path $\pi \in \Pi(q, q')$ and for any $(q, x) \xrightarrow{\pi} (q', x')$, there exists $(q, x) \xrightarrow{\pi_0} (q_1, x_1), (q'_1, x'_1) \xrightarrow{\pi_1} (q_2, x_2), (q'_2, x'_2) \xrightarrow{\pi_2} (q_3, x_3), (q'_3, x'_3) \xrightarrow{\pi_3} (q', x')$ such that the intermediate vectors are not in G and such that $((q_1, x_1), (q'_1, x'_1))$ and $((q_3, x_3), (q'_3, x'_3))$ are in $Id \cup \mathcal{R}^*|_{F_{\infty} \times F_{\infty}}$ and such that $((q_2, x_2), (q'_2, x'_2)) \in Id \cup \mathcal{R}^*|_{F \times F}$. Therefore, we have proved the "flatness witness" inclusion given above.

This concludes the proof that V is flat. We have just proved the following theorem.

Theorem 5.9. Every 2-dim VASS is flat.

Corollary 5.10. The global reachability relation \mathcal{R}^*_V of any 2-dim VASS V is effectively semilinear.

The generic semi-algorithm implemented in the accelerated symbolic model checker FAST is able to compute the reachability set of 40 practical VASS [BFLP03]. Theorem 5.9 shows that this model checker, which was designed to *often* compute the reachability set of practical VASS, also provides a generic algorithm that *always* computes the reachability relation of any 2-dim VASS.

References

[ABS01]	A. Annichini, A. Bouajjani, and M. Sighireanu. TReX: A tool for reach- ability analysis of complex systems. In <i>Proc. 13th Int. Conf. Computer</i> <i>Aided Verification (CAV'2001), Paris, France, July 2001,</i> volume 2102
[BFLP03]	of <i>Lecture Notes in Computer Science</i> , pages 368–372. Springer, 2001. S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Fast Acceleration of Symbolic Transition systems. In <i>Proc. 15th Int. Conf. Computer Aided</i>
	<i>Verification (CAV 2003), Boulder, CO, USA, July 2003,</i> volume 2725 of <i>Lecture Notes in Computer Science,</i> pages 118–121. Springer, 2003.
[BGWW97]	B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In <i>Proc. Static Analysis 4th Int. Symp. (SAS'97), Paris, France,</i>
	Sep. 1997, volume 1302 of Lecture Notes in Computer Science, pages 172–186. Springer, 1997.
[BH99]	A. Bouajjani and P. Habermehl. Symbolic reachability analysis of FIFO- channel systems with nonregular sets of configurations. <i>Theoretical Com-</i> <i>puter Science</i> , 221(1–2):211–250, 1999.
[BJNT00]	A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In <i>Proc. 12th Int. Conf. Computer Aided Verification</i> (<i>CAV'2000</i>), <i>Chicago, IL, USA, July 2000</i> , volume 1855 of <i>Lecture Notes</i>
[BLW03]	in Computer Science, pages 403–418. Springer, 2000. B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large. In Proc. 15th Int. Conf. Computer Aided Verification (CAV'2003), Boul- der, CO, USA, July 2003, volume 2725 of Lecture Notes in Computer
	Science, pages 223–235. Springer, 2003.

- [BM99] A. Bouajjani and R. Mayr. Model checking lossy vector addition systems. In Proc. 16th Ann. Symp. Theoretical Aspects of Computer Science (STACS'99), Trier, Germany, Mar. 1999, volume 1563 of Lecture Notes in Computer Science, pages 323–333. Springer, 1999.
- [BW94] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In Proc. 6th Int. Conf. Computer Aided Verification (CAV'94), Stanford, CA, USA, June 1994, volume 818 of Lecture Notes in Computer Science, pages 55–67. Springer, 1994.
- [CJ98] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. In Proc. 10th Int. Conf. Computer Aided Verification (CAV'98), Vancouver, BC, Canada, June-July 1998, volume 1427 of Lecture Notes in Computer Science, pages 268–279. Springer, 1998.
- [FIS03] A. Finkel, S. P. Iyer, and G. Sutre. Well-abstracted transition systems: Application to FIFO automata. *Information and Computation*, 181(1):1– 31, 2003.
- [FL02] A. Finkel and J. Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In Proc. 22nd Conf. Found. of Software Technology and Theor. Comp. Sci. (FST&TCS'2002), Kanpur, India, Dec. 2002, volume 2556 of Lecture Notes in Computer Science, pages 145–156. Springer, 2002.
- [FS00a] A. Finkel and G. Sutre. An algorithm constructing the semilinear post* for 2-dim reset/transfer vass. In Proc. 25th Int. Symp. Math. Found. Comp. Sci. (MFCS'2000), Bratislava, Slovakia, Aug. 2000, volume 1893 of Lecture Notes in Computer Science, pages 353–362. Springer, 2000.
- [FS00b] A. Finkel and G. Sutre. Decidability of reachability problems for classes of two counters automata. In Proc. 17th Ann. Symp. Theoretical Aspects of Computer Science (STACS'2000), Lille, France, Feb. 2000, volume 1770 of Lecture Notes in Computer Science, pages 346–357. Springer, 2000.
- [GS66] S. Ginsburg and E. H. Spanier. Semigroups, Presburger formulas and languages. *Pacific J. Math.*, 16(2):285–296, 1966.
- [HM00] T. A. Henzinger and R. Majumdar. A classification of symbolic transition systems. In Proc. 17th Ann. Symp. Theoretical Aspects of Computer Science (STACS'2000), Lille, Prance, Feb. 2000, volume 1770 of Lecture Notes in Computer Science, pages 13–34. Springer, 2000.
- [HP79] J. E. Hopcroft and J.-J. Pansiot. On the reachability problem for 5dimensional vector addition systems. *Theoretical Computer Science*, 8(2):135–159, 1979.
- [Kos82] S. R. Kosaraju. Decidability of reachability in vector addition systems. In Proc. 14th ACM Symp. Theory of Computing (STOC'82), San Francisco, CA, May 1982, pages 267–281, 1982.
- [Las] LASH homepage. http://www.montefiore.ulg.ac.be/~boigelot/ research/lash/.
- [May84] E. W. Mayr. An algorithm for the general Petri net reachability problem. *SIAM J. Comput.*, 13(3):441–460, 1984.
- [Min67] M. L. Minsky. *Computation: Finite and Infinite Machines*. Prentice Hall, London, 1 edition, 1967.

Compiling Pattern Matching in Join-Patterns

Qin Ma and Luc Maranget

INRIA-Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France {Qin.Ma, Luc.Maranget}@inria.fr

Abstract. We propose an extension of the join-calculus with pattern matching on algebraic data types. Our initial motivation is twofold: to provide an intuitive semantics of the interaction between concurrency and pattern matching; to define a practical compilation scheme from extended join-definitions into ordinary ones plus ML pattern matching. To assess the correctness of our compilation scheme, we develop a theory of the applied join-calculus, a calculus with value-passing and value matching.

1 Introduction

The join-calculus [5] is a process calculus in the tradition of the π -calculus of Milner, Parrow and Walker [16]. One distinctive feature of join-calculus is the simultaneous definition of all receptors on several channels through *join-definitions*. A join-definition is structured as a list of *reaction rules*, with each reaction rule being a pair of one *join-pattern* and one *guarded process*. A join-pattern is in turn a list of channel names (with formal arguments), specifying the synchronization among those channels: namely, a join-pattern is matched only if there are messages present on all its channels. Finally, the reaction rules of one join-definition define competing behaviors with a non-deterministic choice of which guarded process to fire when several join-patterns are satisfied.

In this paper, we extend the matching mechanism of join-patterns, such that *message* contents are also taken into account. As an example, let us consider the following list-based implementation of a concurrent stack:¹

- def pop(r) & $State(x::xs) \triangleright r(x)$ & State(xs)or push(v) & $State(ls) \triangleright State(v::ls)$
 - in *State*([]) & ...

The second join-pattern push(v) & State(ls) is an *ordinary* one: it is matched whenever there are messages on both *State* and *push*. By contrast, the first joinpattern is an *extended* one, where the formal argument of channel *State* is an *(algebraic) pattern*, matched only by messages that are cons cells. Thus, when the stack is empty *(i.e., when message [] is pending on channel State)*, pop requests are delayed.

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 417-431, 2004.

© Springer-Verlag Berlin Heidelberg 2004

¹ We use the Objective Caml syntax for lists, with *nil* being [] and *cons* being the infix : :.
Note that we follow the convention that capitalized channels are private: only *push* and *pop* will be visible outside.

A similar stack can be implemented without extended join-patterns, by using an extra private channel and ML pattern matching in guarded processes instead:

def pop(r) & $Some(ls) \triangleright match ls$ with $| [x] \rightarrow r(x) \& Empty()$ $| y::x::xs \rightarrow r(y) \& Some(x::xs)$ or $push(v) \& Empty() \triangleright Some ([v])$ or $push(v) \& Some(ls) \triangleright Some (v::ls)$ in $Empty() \& \dots$

However, the second definition obviously requires more programming effort. Moreover, it is not immediately apparent that messages on *Some* are non-empty lists, and that the ML pattern matching thus never fails.

Join-definitions with (constant) pattern arguments appear informally in functional nets [18]. Here we generalize this idea to full patterns. The new semantics is a smooth extension, since both join-pattern matching and pattern matching rest upon classical substitution (or semi-unification). However, an efficient implementation is more involved. Our idea is to address this issue by transforming programs whose definitions contain extended join-patterns into equivalent programs whose definitions use ordinary join-patterns and whose guarded processes use ML pattern matching. Doing so, we leave most of the burden of pattern matching compilation to an ordinary ML pattern matching compiler. However, such a transformation is far more than just straightforward. Namely, there is a gap between (extended) join-pattern matching, which is non-deterministic, and ML pattern matching, which is deterministic (following the "first-match" policy). For example, in our definition of a concurrent stack with extended joinpatterns, State(ls) is still matched by any message on State, regardless of the presence of the more precise State(x::xs) in the competing reaction rule that precedes it. Our solution to this problem relies on partitioning matching values into non-intersecting sets. In the case of our concurrent stack, those sets simply are the singleton $\{[]\}$ and the set of non-empty lists. Then, pattern *State(ls)* is matched by values from both sets, while pattern State(x::xs) is matched only by values of the second set.

The rest of the paper is organized as follows: Section 2 first gives a brief review of patterns and ML pattern matching, then goes on presenting the key ideas of our solution to carry out the transformation informally. We explain how the idea come into being step by step, and especially how we deal with the non-determinism problem. Section 3 presents the semantics of our extended calculus as well as appropriate equivalence relations. Section 4 formalizes the transformation as a compilation scheme and presents the algorithm which essentially works by building a meet semi-lattice of patterns. We go through a complete example in Section 5, and finally, we deal with the correctness of the compilation scheme in Section 6. Most of the proof details are confined to the complementary technical report [14] for lack of space.

2 A Journey Through Patterns

2.1 Patterns and ML Pattern Matching

Patterns and values are built as usual as (well-sorted) terms, over constructor signatures defined by algebraic data types. In contrast to values, patterns may have variables² in them; we require all variables in a pattern to be pairwise distinct. That is, patterns are *linear*. A value v (of type t) is an instance of pattern π (of type t) when there exists a substitution σ , such that $\pi\sigma = v$. In other words, pattern π describes the prefix of instance v, and additionally binds its variables to sub-terms of v. In the following, we write $S(\pi)$ for the set of the instances of pattern π . We have the following relations among patterns (see [11] for full details):

- Pattern π_1 and π_2 are incompatible $(\pi_1 \# \pi_2)$ when $S(\pi_1) \cap S(\pi_2) = \emptyset$.
- Pattern π_1 is less precise then pattern π_2 ($\pi_1 \preceq \pi_2$) when $S(\pi_2) \subseteq S(\pi_1)$.
- Patterns π_1 and π_2 are compatible when they share at least one instance. Two compatible patterns admit a least upper bound (for \leq) written $\pi_1 \uparrow \pi_2$, whose instance set is $S(\pi_1) \cap S(\pi_2)$.
- Patterns π_1 and π_2 are equivalent $(\pi_1 \equiv \pi_2)$ when $S(\pi_1) = S(\pi_2)$. If so, their least upper bound is their representative, written $\pi_i \uparrow \pi_2$.³

ML pattern matching is deterministic, even when patterns are overlapping (*i.e.*, compatible). More precisely, consider the following ML pattern matching

match e with $\mid \pi_1 \rightarrow Q_1 \mid \pi_2 \rightarrow Q_2 \mid \ldots \mid \pi_n \rightarrow Q_n$

Pattern π_i is matched by the values in set $S(\pi_i) \setminus (\bigcup_{1 \le j < i} S(\pi_j))$ and only by those. In other words, given some value v, patterns $\pi_1, \pi_2, \ldots, \pi_n$ are checked for having v as an instance, in that order, stopping as soon as a match is found, pattern matching is exhaustive when $\bigcup_{1 \le i \le n} S(\pi_j)$ is the whole set of values (of the considered type).

2.2 Transform Pattern Arguments in Join to ML Pattern Matching

The implementation of extended join-synchronization requires to test message contents against pattern arguments, while ordinary join-synchronization only requires to test message presence. Our idea is to separate algebraic pattern testing from join-synchronization, and to perform the former operation by using ML-pattern matching. To avoid inappropriate message consumption, message contents are tested first. Let's consider the following join-definition:

def
$$c(\pi_1) \& d(...) \triangleright P_1$$

or $c(\pi_2) \& e(...) \triangleright P_2$

We refine channel *c* into more precise ones, each of which carries the instances of patterns π_1 or π_2 .

² In patterns, we freely replace variables whose names are irrelevant by *wild-cards* "_".

³ Because of typing, there exists non-trivial equivalences such as at any pair type, we have $_ \equiv (_,_)$.

def $c_{\pi_1}(\ldots)$ & $d(\ldots) \triangleright P_1$ or $c_{\pi_2}(\ldots)$ & $e(\ldots) \triangleright P_2$

Then, we add a new reaction rule to dispatch the messages on channel c to either c_{π_1} or c_{π_2} :

or
$$c(v) \triangleright$$
 match v with
 $\mid \pi_1 \rightarrow c_{\pi_1}(...)$
 $\mid \pi_2 \rightarrow c_{\pi_2}(...)$
 $\mid _ \rightarrow \emptyset$

The notation \emptyset stands for the null process, which is used in the last matching rule to discard messages that match neither π_1 nor π_2 .

The simple compilation above works perfectly, as long as π_1 and π_2 are incompatible. Unfortunately, it falls short when π_1 and π_2 have common instances, namely, the non-determinism problem. However, further refinements can handle this situation.

- If $\pi_1 \preceq \pi_2$, (but $\pi_2 \not\preceq \pi_1$), that is if all instances of π_2 are instances of π_1 , then, to get a chance of meeting its instances, pattern π_2 must come first:

or
$$c(v) \triangleright$$
 match v with
 $\mid \pi_2 \rightarrow c_{\pi_2}(...)$
 $\mid \pi_1 \rightarrow c_{\pi_1}(...)$
 $\mid - \rightarrow \emptyset$

But now, channel c_{π_1} does not carry all the possible instances of pattern π_1 anymore, instances shared by pattern π_2 are dispatched to c_{π_2} . As a consequence, the actual transformation of the initial reaction rules is as follows:

def $c_{\pi_1}(\ldots)$ & $d(\ldots) \triangleright P_1$ or $c_{\pi_2}(\ldots)$ & $d(\ldots) \triangleright P_1$ or $c_{\pi_2}(\ldots)$ & $e(\ldots) \triangleright P_2$

Observe that non-determinism is now more explicit: an instance of π_2 sent on channel *c* can be consumed by either reaction rule. We can shorten the new definition a little by using or in join-patterns:

def
$$(c_{\pi_1}(\ldots) \text{ or } c_{\pi_2}(\ldots)) \& d(\ldots) \triangleright P_1$$

or $c_{\pi_2}(\ldots) \& e(\ldots) \triangleright P_2$

- If $\pi_1 \equiv \pi_2$, then matching by their representative is enough:

def $c_{\pi_1 \uparrow \pi_2}(\ldots) \& d(\ldots) \triangleright P_1$ or $c_{\pi_1 \uparrow \pi_2}(\ldots) \& e(\ldots) \triangleright P_2$ or $c(v) \triangleright$ match v with $\mid \pi_1 \uparrow \pi_2 \rightarrow c_{\pi_1 \uparrow \pi_2}(\ldots)$ $\mid - \rightarrow \emptyset$

- Finally, if neither $\pi_1 \preceq \pi_2$ nor $\pi_2 \preceq \pi_1$ holds, with π_1 and π_2 being nevertheless compatible, then an extra matching by pattern $\pi_1 \uparrow \pi_2$ is needed:

def
$$(c_{\pi_1}(\ldots) \text{ or } c_{\pi_1 \uparrow \pi_2}(\ldots)) \& d(\ldots) \triangleright P_1$$

or $(c_{\pi_2}(\ldots) \text{ or } c_{\pi_1 \uparrow \pi_2}(\ldots)) \& e(\ldots) \triangleright P_2$
or $c(v) \triangleright \text{ match } v \text{ with}$

$$\begin{array}{c|c} & \pi_1 \uparrow \pi_2 \to c_{\pi_1 \uparrow \pi_2} (\ldots) \\ & & \pi_1 \to c_{\pi_1} (\ldots) \\ & & \pi_2 \to c_{\pi_2} (\ldots) \\ & & & \downarrow _ \to \emptyset \end{array}$$

Note that the relative order of π_1 and π_2 is irrelevant here.

In the transformation rules above, we paid little attention to variables in patterns, by writing $c_{\pi}(...)$. We now demonstrate variable management by means of our stack example. Here, the relevant patterns are $\pi_1 = \ell$ and $\pi_2 = x: :xs$ and we are in the case where $\pi_1 \leq \pi_2$ (and $\pi_2 \not\leq \pi_1$ because of instance []). Our idea is to let dispatching focus on instance checking, and to perform variable binding after synchronization:

def
$$pop(r)$$
 & $State_{x::xs}(z) \triangleright match z$ with $x::xs \rightarrow r(x)$ & $State(xs)$
or $push(v)$ & $(State_{x::xs}(z) \text{ or } State_{ls}(z)) \triangleright match z$ with $ls \rightarrow State(v::ls)$
or $State(v) \triangleright match v$ with
 $| x::xs \rightarrow State_{x::xs}(v) |$
 $| ls \rightarrow State_{ls}(v)$

One may believe that the matching of the pattern x::xs needs to be performed twice, but it is not necessary. The compiler in fact knows that the matching of z against x::xs (on first line) cannot fail. As a consequence, no test needs to be performed here, only the binding of the pattern variables. Moreover, the existing optimizing pattern matching compiler of [11] can be fooled into producing minimal code for such a situation by simply asserting that the compiled matching is exhaustive.

3 The Applied Join-Calculus

In this section, we define the applied join-calculus by analogy with "the applied π -calculus" [1]. The applied join-calculus inherits its capabilities of communication and concurrency from the pure join-calculus [5]. Moreover it extends to support algebraic value-passing, and algebraic pattern matching in both join-patterns and guarded processes.

3.1 Syntax and Scopes

The syntax of the applied join-calculus is given in Figure 1. Constructors of algebraic data types have an arity and are ranged over by C. A constructor with arity 0 is a constant. We assume an infinite set of variables, ranged over by a, b, \ldots, y, z .

Two new syntactic categories are introduced: expressions and patterns. At the first glance, both expressions and patterns are terms constructed from variables and constructors, where n matches the arity of constructor C. However, we require patterns to be linear. ML pattern matching is added as a process, which matches the value of the expression against a list of patterns. Moreover, in contrast to ordinary name-passing join-calculus, there are two more radical

P ::=Processes Ø null process x(e)message sending P & Pparallel def D in Pdefinition match e with $|\pi_1 \to P_1| \dots |\pi_m \to P_m$ ML pattern matching D ::=Join-definitions Т empty definition $J \triangleright P$ reaction D or Ddisjunction J ::=Join-patterns $x(\pi)$ message pattern J & Jsynchronization $\pi ::=$ Patterns variable $C(\pi_1,\pi_2,\ldots,\pi_n)$ constructor pattern e ::= Expressions variable $C(e_1, e_2, \ldots, e_n)$ constructor expression

Fig. 1. Syntax of the applied join-calculus

extensions: first, message contents become expressions, that is, we have valuepassing; second, when a channel name is defined in a join-pattern, we also specify what pattern the message content should satisfy.

There are two kinds of bindings: the definition process def D in P binds all the channel names defined in D (dn[D]) in the scope of P; while the reaction rule $J \triangleright P$ or the ML pattern matching match e with $|\pi_1 \rightarrow P_1| \dots |\pi_m \rightarrow P_m$ bind all the local variables (rv[J] or rv[π_i]) in the scope of P or P_i , $i \in \{1, \dots, m\}$.

The definitions of the set of defined channel names $dn[\cdot]$, the set of local variables $rv[\cdot]$, and the set of free variables $fv[\cdot]$ are almost the same as in the join-calculus, except for the following modifications or extensions to adopt patterns.

$$\begin{aligned} \operatorname{rv}[c(\pi)] &\stackrel{\text{def}}{=} \operatorname{rv}[\pi] \\ \operatorname{rv}[x] &\stackrel{\text{def}}{=} \{x\} \\ \operatorname{rv}[C(\pi_1, \pi_2, \dots, \pi_n)] &\stackrel{\text{def}}{=} \operatorname{rv}[\pi_1] \uplus \operatorname{rv}[\pi_2] \uplus \dots \uplus \operatorname{rv}[\pi_n] \\ \end{aligned}$$
$$\begin{aligned} \operatorname{fv}[\operatorname{match} u \text{ with } |^{i \in I} \pi_i \to P_i] \\ &\stackrel{\text{def}}{=} \operatorname{fv}[u] \cup (\bigcup_{i \in I} \operatorname{fv}[P_i] \setminus \operatorname{fv}[\pi_i]) \end{aligned}$$

We assume a type discipline in the style of the type system of the joincalculus [7], extended with constructor types and the rule for ML pattern matching. Without making the type discipline more explicit, we consider only welltyped terms (whose type we know), and assume that substitutions preserve types. It should be observed that the arity checking of polyadic join-calculus is replaced by a well-typing assumption in our calculus, which is monadic and

STR-NULL	$\vdash \emptyset \rightleftharpoons \vdash$
Str-Par	$\vdash P \& P' \rightleftharpoons \vdash P, P'$
Str-Top	$\top \vdash \dashv \vdash$
Str-And	$D ext{ or } D' \vdash \ \rightleftharpoons \ D, D' \vdash$
STR-DEF	$\vdash def \ D \ in \ P \ \rightleftharpoons \ D \vdash P$
React	$J \triangleright P \vdash J\sigma \longrightarrow J \triangleright P \vdash P\sigma$
Match	\vdash match $\pi_i \rho$ with $ \pi_1 \rightarrow P_1 \dots \pi_m \rightarrow P_m $
	$\longrightarrow \vdash P_i \rho$

Side conditions:

STR-DEF	dn[D] is fresh
React	σ substitutes (closed) expressions for $rv[J]$
MATCH	ρ substitutes (closed) expressions for $rv[\pi_i]$ and $\forall j < i, \pi_j \not\preceq \pi_i \rho$

Fig. 2. RCHAM of the applied join-calculus

whose message contents can be tuples. However, one important consequence of typing is that any (free) variable in a term possesses a type and that we know this type. Hence, we can discriminate between those variables that are of a type of constructed values and those that are of channel type. Generally speaking, in name-passing calculi semantics, the latter kind of variables are (almost) treated as channel names, that is, values. While, in any reasonable semantics, the former kind of variables cannot be treated so. Reduction will operate on *variable-closed* (*closed* for short) terms, whose free variables are all of channel type.

Finally, we use the or construct in join-patterns as syntax sugar, in the following sense:

$$J \& (J_1 \text{ or } J_2) \triangleright P = (J \& J_1 \triangleright P) \text{ or } (J \& J_2 \triangleright P)$$

3.2 Reduction Semantics

We establish the semantics in the reflexive chemical abstract machine style [5, 3]. A *chemical solution* is a pair $\mathcal{D} \vdash \mathcal{P}$, where \mathcal{D} is a multiset of join-definitions, and \mathcal{P} is a multiset of processes. Extending the notion of closeness to solutions, a solution is closed when all the join-definitions and processes in it are closed. The chemical rewrite rules are given in Figure 2. They apply to closed solutions, and consist of two kinds: structural rules \rightarrow or \neg represent the syntactical rearrangement of the terms, and reduction rules \rightarrow represent the computation steps. We follow the convention to omit the part of the solution which remains unchanged during rewrite.

Matching of message contents by formal arguments is integrated in the substitution σ in rule REACT. As a consequence this rule does not formally change with respect to ordinary join-calculus. However its semantical power has much increased. The MATCH rule is new and expresses ML pattern matching.

According to the convention of processes as solutions, namely P as $\vdash P$, the semantics is also defined on closed processes in the following sense.

Definition 1. Denote \rightleftharpoons^* as the transitive closure of $\rightharpoonup \cup \neg$,

1. $P \equiv Q$ iff $\vdash P \rightleftharpoons^* \vdash Q$ 2. $P \longrightarrow Q$ iff $\vdash P \rightleftharpoons^* \longrightarrow \rightleftharpoons^* \vdash Q$

Obviously, we have the structural rule, namely, if $P \longrightarrow Q$, $P \equiv P'$, and $Q \equiv Q'$, then $P' \longrightarrow Q'$.

3.3 Equivalence Relation

In this section, we equip the applied join-calculus with equivalence relations to allow equational reasoning among processes. The classical notion of *barbed congruence* is a sensible behavioral equivalence based on a reduction semantics and barb predicates. It was initially proposed by Milner and Sangiorgi for CCS [17], and adapted to many other process calculi [9,2], including the join-calculus [5]. We take *weak barbed congruence* [17] as our basic notion of "behavioral equivalence" for closed processes.

Definition 2 (Barb Predicates). *Let P be a closed process, and c be a channel name*

- 1. P has a strong barb on c: $P \downarrow_c$, iff $P \equiv (\det D \operatorname{in} Q) \& c(e)$, for some D, Q and e.
- 2. P has a weak barb on channel $c: P \Downarrow_c$, iff $P \longrightarrow^* P'$ such that $P' \downarrow_c$.

Definition 3 (Weak Barbed Bisimulation). A binary relation \mathcal{R} on closed processes is a weak barbed bisimulation if, whenever PRQ, we have

1. If $P \longrightarrow P'$, then $\exists Q'$, s.t. $Q \longrightarrow^* Q'$ and $P'\mathcal{R}Q'$, and vice versa. 2. For any $c, P \Downarrow_c$ iff $Q \Downarrow_c$.

By definition, Weak barbed bisimilarity ($\dot{\approx}$) is the largest weak barbed bisimulation.

A *context* $C[\cdot]$ is a term built by the grammar of process with a single process placeholder $[\cdot]$. An *executive contexts* $E[\cdot]$ is a context in which the placeholder is not guarded. Namely:

 $E[\cdot] \stackrel{\text{def}}{=} [\cdot] \mid E[\cdot] \& P \mid P \& E[\cdot] \mid \text{def } D \text{ in } E[\cdot]$

We say a context is closed if all the free variables in it are of channel type.

Definition 4 (Weak Barbed Congruence). A binary relation on closed processes is a weak barbed congruence if it is a weak barbed bisimulation and closed by application of any closed executive context. We denote the largest weak barbed congruence as \approx .

The weak barbed congruence \approx is defined on the closed subset of the applied join-calculus. Although the definition itself only requires the closure of executive contexts, it can be proved that the full congruence does not provide

more discrimination power. Similarly to what Fournet has established for the pure join-calculus in his thesis [4], we first have the property that \approx is closed under substitution because, roughly, name substitutions may be mimicked by executive contexts with "forwarders".

Lemma 1. Given two closed processes P and Q, if $P \approx Q$, then for any substitution σ , $P\sigma \approx Q\sigma$. (Note that "closed" stands for "variable-closed".)

Then based on this property, the full congruence is also guaranteed.

Theorem 1. Weak barbed congruence \approx is closed under application of any closed context.

Up to now, we define the weak barbed congruence as expressing the equivalence of two processes at runtime. However, this is not sufficient for reasoning the behavior of our compilation, which applies perfectly well to processes with free variables. In other words, we also need a way to express the equivalence of two processes statically. Of course, the static equivalence must imply the runtime one. Therefore, the equivalence relation of any processes, whether closed or not, is defined in terms of the runtime equivalence relation \approx , using substitutions to close up.

Definition 5 (Static Equivalence). Two processes P and Q are statically equivalent (\Rightarrow), if for any substitution σ such that $P\sigma$ and $Q\sigma$ are closed, $P\sigma \approx Q\sigma$.

Following the definition, we can check that \Rightarrow is closed by substitution.

Lemma 2. $P \Rightarrow Q \implies \forall \sigma. P \sigma \Rightarrow Q \sigma$.

More importantly, \Rightarrow is also closed with respect to all contexts. Namely, the following theorem holds.

Theorem 2. The static equivalence *⇒* is a full congruence.

There is still a good property worth noticing: in fact, for the closed subset of the applied join-calculus, we have \Rightarrow and \approx coincide. This is almost straightforward following the definition of static equivalence and Lemma 1.

4 The Compilation [.]

We formalize the intuitive idea described in Section 2 as a transformer Y_c , which transforms a join-definition *w.r.t.* channel *c.* The algorithm essentially works by constructing the meet semi-lattice of the formal pattern arguments of channel *c* in *D*, modulo pattern equivalence \equiv , and with relation \preceq as partial order. Moreover, we visualize the lattice as a Directed Acyclic Graph, namely, vertices as patterns, and edges representing the partial order. If we reason more on instance sets than on patterns, this structure is quite close to the "subset graph" of [20].

Algorithm Y_c : Given D, the join-definition to be transformed.

Step 0: Pre-process

- 1. Collect all the pattern arguments of channel c into the sequence: $\Pi = \pi_1; \pi_2; \ldots; \pi_n$.
- 2. Let Π' be formed from Π by taking the \uparrow of all equivalent patterns; thus Π' is a sequence of pairwise non-equivalent patterns.
- 3. Perform exhaustiveness check on Π' , if not exhaustive, issue a warning.
- 4. IF There is only one pattern in Π' , and that Π' is exhaustive **THEN** goto Step 5 (In that case, no dispatching is needed.)

Step 1: Closure of Least Upper Bound

For any pattern γ and pattern sequence $X = \gamma_1; \gamma_2; \ldots; \gamma_n$, we define $\gamma \uparrow X$ as the sequence $\gamma \uparrow \gamma_{i_1}; \gamma \uparrow \gamma_{i_2}; \ldots; \gamma \uparrow \gamma_{i_m}$, where the γ_{i_k} 's are the patterns from X that are compatible with γ .

We also define function F, which takes a pattern sequence X as argument and returns a pattern sequence.

IF X is empty

THEN F(X) = X

ELSE Decompose X as γ ; X' and state $F(X) = \gamma$; F(X'); $\gamma \uparrow F(X')$

Compute the sequence $\Gamma = F(\Pi')$. It is worth noticing that Γ is the sequence of valid patterns $(\pi'_{i_1} \uparrow \dots (\pi'_{i_{k-1}} \uparrow \pi'_{i_k}) \dots)$, with $1 \leq i_1 < i_2 < \dots < i_k \leq n'$, and $1 \leq k \leq n'$, where we decompose Π' as $\pi'_1; \pi'_2; \dots; \pi'_{n'}$.

Step 2: Up to Equivalence

As in Step 0.2, build Γ' from Γ .

Step 3: Build DAG

Corresponding to the semi-lattice (Γ', \preceq) , build a directed acyclic graph G(V, E).

1. $V = \emptyset, E = \emptyset$.

- 2. For each pattern γ in Γ' , add a new vertex v into V and labeled the vertex with γ , written as $label(v) = \gamma$.
- ∀(v, v') ∈ V × V, v ≠ v', if label(v) ≤ label(v'), then add an edge from v' to v into E.

Step 4: Add Dispatcher

Following one topological order, the vertices of G are indexed as v_1, \ldots, v_m . We extend the join-definition D with a dispatcher on channel c of the form: $c(x) \triangleright \text{match } x \text{ with } \mathcal{L}$, where x is a fresh variable and \mathcal{L} is built as follows:

- 1. Let j ranges over $\{1, \ldots, m\}$. Following the above topological order, for all vertices v_j in V append a rule "| $label(v_j) \rightarrow c_j(x)$ " to \mathcal{L} , where c_j is a fresh channel name.
- 2. If Π' is not exhaustive, then add a rule " $_ \rightarrow \emptyset$ " at the end.

Step 5: Rewrite Reaction Rules

For each reaction rule defining channel c in D: $J_i \& c(\pi_i) \triangleright Q_i$, we rewrite it according to the following policy. Let $Q'_i = \text{match } x_i \text{ with } \pi_i \to Q_i$, where x_i is a fresh variable.

IF coming from Step 0 **THEN** rewrite to $J_i \& c(x_i) \triangleright Q'_i$ **ELSE**

- 1. Let v_{j_i} be the unique vertex in V, s.t. $label(v_{j_i}) \equiv \pi_i$.
- 2. We collect all the predecessors of v_{j_i} in *G*, and we record the indexes of them, together with j_i , into a set notated as $I(\pi_i)$.
- 3. Rewrite to $J_i \& (\bigvee_{j \in I(\pi_i)} c_j(x_i)) \triangleright Q'_i$, where \bigvee is the generalized or construct of join-patterns.

Assume $dn[D] = \{c_1, \ldots, c_n\}$ $(n \ge 0)$, where we assume an order on the channel names. To transform a join-definitions D, we just apply $Y_{c_n} \ldots Y_{c_1}(D)$. And the compilation of processes $\llbracket \cdot \rrbracket$ is defined as follows.

$$\begin{split} \llbracket \emptyset \rrbracket \stackrel{\text{def}}{=} \emptyset \\ \llbracket x(e) \rrbracket \stackrel{\text{def}}{=} x(e) \\ \llbracket P \& P' \rrbracket \stackrel{\text{def}}{=} \llbracket P \rrbracket \& \llbracket P' \rrbracket \\ \llbracket \text{def } D \text{ in } P \rrbracket \stackrel{\text{def}}{=} \text{def } Y_{c_n} \dots Y_{c_1}(D) \text{ in } \llbracket P \rrbracket \\ \llbracket \text{match } e \text{ with } |^{i \in I} \pi_i \to P_i \rrbracket \stackrel{\text{def}}{=} \text{match } e \text{ with } |^{i \in I} \pi_i \to \llbracket P_i \rrbracket \end{split}$$

Observe that the compilation preserves the interface of join-definitions. Namely, it only affects definitions *D*, while message sending remains the same.

5 Example of Compilation

Given the following join-definition of an enriched integer stack

 $\begin{array}{l} \text{def } push(v) \& State(ls) \triangleright State \; (v::ls) \\ \text{or } pop(r) \& State(x::xs) \triangleright r(x) \& State(xs) \\ \text{or } insert(n) \& State(0::xs) \triangleright State(0::n::xs) \\ \text{or } last(r) \& State([x]) \triangleright r(x) \& State([x]) \\ \text{or } swap() \& State(x_1::x_2::xs) \triangleright State(x_2::x_1::xs) \\ \text{or } pause(r) \& State([]) \triangleright r() \\ \text{or } resume(r) \triangleright State([]) \& r() \end{array}$

The *insert* channel inserts an integer as the second topmost element, but only when the topmost element is 0. The *last* channel gives back the last element in the stack, keeping the stack unchanged. The *swap* channel exchange the topmost two elements in the stack. The *pause* channel temporarily freezes the stack when it is empty, while the *resume* channel brings the stack back into work. We now demonstrate our transformation *w.r.t.* channel *State*.

Step 0. We collect the pattern arguments of channel *State* into Π

$$\Pi = ls; \ x::xs; \ 0::xs; \ [x]; \ x_1::x_2::xs; \ []$$

Because none of these patterns is equivalent to another, $\Pi' = \Pi$. Additionally, Π' is exhaustive (pattern *ls* alone covers all possibilities).

Step 1,2. Γ extends Π' with all possible least upper bounds. Then we form Γ' from Γ by taking the \uparrow of all equivalent patterns.

 $\Gamma' = ls; x::xs; 0::xs; [x]; x_1::x_2::xs; []; 0::x_2::xs; [0]$

TEAM LING

Note that the last two patterns are new, where

$$\begin{array}{rcl} 0::x_{2}::xs &=& 0::xs \uparrow x_{1}::x_{2}::xs \\ & & [0] &=& 0::xs \uparrow [x] \end{array}$$

Step 3. We build the semi-lattice (Γ', \preceq) , see Figure 3.

Step 4. One possible topological order of the vertices is also given at the right

of Figure 3. Following that order, we build the dispatcher on channel State.

or $State(y) \triangleright$ match y with

```
|\begin{array}{c} 0::x_2::x_5 \rightarrow State_1(y) \\ | \begin{array}{c} [0] \rightarrow State_2(y) \\ | \begin{array}{c} x_1::x_2::x_5 \rightarrow State_3(y) \\ | \begin{array}{c} 0::x_5 \rightarrow State_4(y) \\ | \begin{array}{c} [x] \rightarrow State_5(y) \\ | \begin{array}{c} x::x_5 \rightarrow State_6(y) \\ | \begin{array}{c} [] \rightarrow State_7(y) \\ | \begin{array}{c} l_5 \rightarrow State_8(y) \end{array} \\ | \begin{array}{c} l_5 \rightarrow State_8(y) \end{array}
```

Step 5. We rewrite the original reaction rules. As an example, consider the third reaction rule for the *insert* behavior: the pattern in *State*(0:: *xs*) corresponds to vertex 4 in the graph, which has two predecessors: vertex 1 and vertex 2. Therefore, the reaction rule is rewritten to

 $insert(n) \& (State_1(x_3) \text{ or } State_2(x_3) \text{ or } State_4(x_3))$ > match x_3 with $0::xs \rightarrow State(0::n::xs)$

where $State_1$, $State_2$ and $State_4$ are the fresh channel names corresponding to vertices 1, 2, 4 respectively, and x_3 is a fresh variable.

As a final result of our transformation, we get the disjunction of the following rules and of the dispatcher built in Step 4.

As discussed at the end of Section 2, ML pattern matchings in the guarded processes are here only for binding pattern variables. Therefore, if the original pattern does not contain any variables (c.f. the *pause* rule), we can discard the ML pattern matching, as shown in the above program.



Fig. 3. The semi-lattice of patterns and the topological order

6 Correctness

A program written in the extended join-calculus of Section 3 is a process *P*. The compilation [P] replaces all the join-definitions *D* in *P* by $Y_{c_n} \ldots Y_{c_1}$ (*D*), where $dn[D] = \{c_1, \ldots, c_n\}$. To guarantee the correctness, we require the programs before and after the compilation to be statically equivalent. Namely, the following theorem should hold.

Theorem 3. For any process P, $\llbracket P \rrbracket \Leftrightarrow P$.

Proof. By structural induction on processes, and because \Rightarrow is a full congruence and a transitive relation, it suffices to prove the following Lemma 3.

Lemma 3. For any join-definition D, channel name $c \in dn[D]$, and process P,

def D in $P \Leftrightarrow def Y_c(D)$ in P

This lemma is crucial to the correctness of the compilation. The proof relies on the properties of the meet semi-lattice constructed from the pattern arguments. In particular the proof exploits the deterministic semantics of the ML pattern matching in the dispatcher, which is built following the topological order of the lattice. For lack of space, we omit the proof. Please refer to the complementary technical report.

7 Conclusion and Future Work

In this paper we have introduced the applied join-calculus. The applied joincalculus inherits its capabilities of communication and concurrency from the pure join-calculus and supports value-passing. The one significant extension lies in providing the power of pattern matching. Thus, the applied join-calculus is a more precise and realistic model combining both functional and concurrent programming.

Our calculus is thus "impure" in the sense of Abadi and Fournet's applied π -calculus [1]. We too extend an archetypal name-passing calculus with pragmatic

constructs, in order to provide a full semantics that handles realistic language features without cumbersome encodings. It is worth noticing that like in [1], we distinguish between variables and names, a distinction that is seldom made in pure calculi. Since we aim to prove a program transformation correct, we define the equivalence on open terms, those which contain free variables. Abadi and Fournet are able to require their terms to have no free variables, since their goal is to prove properties of program execution (namely the correctness of security protocols).

Our compilation scheme can be seen as the combination of two basic steps: dispatching and forwarding. The design and correctness of the dispatcher essentially stems from pattern matching theory, while inserting an internal forwarding step in communications is a natural idea, which intuitively does not change process behavior. Various works give formal treatments of the intuitive correctness of forwarders, in contexts different from ours. For instance, forwarders occur in models of concrete distribution in the π -calculus [15, 8]. Of course, our interest in forwarders has quite different motivations. In particular, our dispatcher may forward messages on several channels, taking message contents into account, thereby performing some demultiplexing. However, the proof techniques and objective (which can be summarized as "full abstraction") are quite similar.

As regards implementation, we claim that our transformation can be integrated easily in the current JoCaml system [10]. The JoCaml system is built on top of Objective Caml [12], a dialect of ML, which features a sophisticated ML pattern matching compiler [11]. Our transformation naturally takes place between the typing and ML pattern matching compilation phases of the existing compiler. More significantly, this should be the only addition. In particular, our solution does not require any modification of the existing runtime system since the join-pattern synchronization remains as before. It is worth observing that a direct implementation of extended join-pattern matching at the runtime level would significantly complicate the management of message queues, which would then need to be scanned in search of matching messages before consuming them.

The integration of pattern matching into the join-calculus is part of our effort to develop a practical concurrent programming language with firm semantical foundations (a similar effort is for instance Scala [19]). In our opinion, a programming language is more than an accumulation of features. That is, features interact sometimes in unexpected ways, especially when intimately entwined. Here, we introduce algebraic patterns as formal arguments of channel definitions. Doing so, we provide a more convenient (or "expressive") language to programmers. From that perspective, pattern matching and join-calculus appear to live well together, with mutual benefits.

In previous work, we have designed an object-oriented extension of the joincalculus [6, 13], which appeared to be more difficult. The difficulties reside in the refinement of the synchronization behavior of objects by using the inheritance paradigm. We solved the problem by designing a delicate way of rewriting joinpatterns at the class level. However, the introduction of algebraic patterns in join-patterns impacts this class-rewriting mechanism. The interaction is not immediately clear. Up to now, we are aware of no object-oriented language where the formal arguments of methods can be patterns. We thus plan to investigate such a combination of pattern matching and inheritance, both at the calculus and language level.

Acknowledgements. The authors wish to thank James Leifer and Jean-Jacques Lévy for fruitful discussions and comments.

References

- 1. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proceedings of POPL'01*, pages 104–115, 2001.
- 2. R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π -calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.
- 3. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.
- 4. C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming.* PhD thesis, Ecole Polytechnique, Nov. 1998.
- 5. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL'96*, pages 372–385, 1996.
- 6. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the join calculus. *Journal of Logic and Algebraic Programming*, 57(1-2):23–69, 2003.
- C. Fournet, L. Maranget, C. Laneve, and D. Rémy. Implicit typing à la ML for the join-calculus. In *Proceedings of CONCUR*'97, LNCS 1243, pages 196–212, 1997.
- P. Gardner, C. Laneve, and L. Wischik. Linear forwarders. In *Proceedings of CONCUR'03*, LNCS 2761, pages 415–430, 2003.
- 9. K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 151(2):437–486, 1995.
- 10. F. Le Fessant. The JoCaml system. http://pauillac.inria.fr/jocaml, 1998.
- 11. F. Le Fessant and L. Maranget. Optimizing pattern-matching. In *Proceedings of ICFP'01*, pages 26–37, 2001.
- 12. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml System, version 3.07. http://caml.inria.fr/, 2003.
- Q. Ma and L. Maranget. Expressive synchronization types for inheritance in the join calculus. In *Proceedings of APLAS'03*, LNCS 2895, pages 20–36, 2003.
- Q. Ma and L. Maranget. Compiling pattern matching in join-patterns. Rapport de recherche 5160, INRIA-Rocquencourt, Apr. 2004. Available at http://pauillac. inria.fr/~ma/papers/ptjoin-tr.ps.
- M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *Proceedings* of *ICALP*'98, LNCS 1443, pages 856–867, 1998.
- R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, 1992.
- 17. R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proceedings of ICALP'92*, volume LNCS 623, pages 685–695, 1992.
- M. Odersky. Functional nets. In *Proceedings of ESOP'00*, LNCS 1782, pages 1–25, 2000.
- 19. M. Odersky. The Scala Language. http://lamp.epfl.ch/~odersky/scala/,2002.
- 20. P. Pritchard. On computing the subset graph of a collection of sets. *Journal of Algorithms*, 33(2):187–203, 1999.

Model Checking Restricted Sets of Timed Paths

Nicolas Markey and Jean-François Raskin

Département d'Informatique Université Libre de Bruxelles Bld du Triomphe – CP 212 1050 Brussels – Belgium {nmarkey, jraskin}@ulb.ac.be

Abstract. In this paper, we study the complexity of model-checking formulas of three important real-time logics (MTL, MITL, and TCTL) over restricted sets of timed paths. The classes of restricted sets of timed paths that we consider are (*i*) a single finite (or ultimately periodic) timed path, (*ii*) a infinite set of finite (or infinite) timed paths defined by a finite (or ultimately periodic) path in a region graph, (*iii*) a infinite set of finite (or ultimately periodic) path in a zone graph.

Introduction

Timed automata have been introduced in [2] as a formal notation to model behaviors of real-time systems. Requirements for real-time systems modeled as timed automata are conveniently expressed using *real-time logics*. Real-time logics are *quantitative extensions* of temporal logics. Three main logics have been defined to express real-time requirements: TCTL [1] is a real-time extension of the CTL logic, while MTL and MITL [3] are extensions of the LTL logic. The model checking problems for those logics over *sets of timed paths* defined by timed automata have been studied. The results are as follows: For the logic TCTL, the model checking problem has been shown PSPACE-complete in [1]. For the logic MITL, the problem has been shown undecidable in [4]. For the logic MITL, the problem has been shown EXPSPACE-complete in [3].

In this paper, we study the model checking problems for those real-time logics on *several classes* of *restricted* sets of timed paths. We consider the model checking problems related to TCTL, MTL, and MITL when the set of timed paths is (*i*) a single finite (or ultimately periodic) timed path, (*ii*) a set of finite (or ultimately periodic) timed path, (*iii*) a set of finite (or infinite) timed paths defined by a finite (or ultimately periodic) path in a region graph, (*iii*) a set of finite (or infinite) timed paths defined by a finite (or ultimately periodic) path in a zone graph. Note that in cases (*ii*) and (*iii*), the sets contain *uncountably many* timed paths. Note also that finite or ultimately periodic region paths as well as zone paths can be seen as *simple* form of timed automata.

Beside the theoretical interest to study the complexity of the model checking problems for those subcases, there are important practical reasons to study them.

First, verification algorithms for timed automata have to manipulate symbolically infinite state spaces. This is done either through the region graph or through the zone graph. When the verification of a *safety* or a *linear-time property* fails, those symbolic algorithms identify a finite or an infinite ultimately periodic path in the region graph or in the zone graph [6]. This path is the symbolic representation of an infinite set of timed paths that are counter-examples to the property. Usually, the information that is given back to the user is a single timed path extracted from this symbolic path. Nevertheless, it may be much more interesting to give to the user not only a single counter-example but the entire infinite set of counter-examples actually computed by the symbolic algorithm. As this counter example is symbolic, the possibility to analyze this counter-example using model checking should be given to the user. In fact, in order to better understand this infinite set of counter examples, the user may want to formulate model checking questions about this set. We should then look if we can specialize our verification algorithms for those (possibly) simpler problems.

Second, a real-time system that is executing constructs naturally a timed path that represent the evolution of its state along time. Correctness requirements about this single execution path can be expressed using a linear real-time logic, like MTL. Can we efficiently verify properties expressed by MTL formulas on this single timed path? In the dense-time framework, we know from the undecidability result for MTL that we cannot construct, as in the finite state case for LTL, a *monitor* (in the form of a timed automaton for example) that will enter a bad state in the case the formula is not satisfied. It is clear again that we need to look at specific techniques.

Third, if a timed automaton is too complex to be completely verified, we may be interested to *test* it instead. Testing a timed automaton against a set of real-time formulas consists in: (i) extracting a set of timed paths out of the timed automaton by executing it, and (ii) verifying that the set of extracted path verify some given real-time formulas. The set of timed paths can be either extracted by explicit execution of the timed automaton (in this case it is a finite set) or, more interestingly, extracted through symbolic execution of the timed automaton which amounts to visit a subset of its region or zone graph. In the two latter cases, we must check real-time formulas on infinite sets of timed paths defined by a finite set of region or zone paths. Again, for those subcases, we should look at the existence of specialized techniques.

The results that we have obtained for the model checking problems of the three real-time logics over the six classes of restricted sets of timed paths are given in Table 1. To the best of our knowledge, only the three results from the first line were known, all the other results are new. The undecidability and EXP-SPACE-hard result for the model checking of MTL and MITL over ultimately periodic region paths were *unexpected* and their proofs have required new encoding techniques for Turing Machine computations using timed paths. Those hardness results imply hardness results for ultimately periodic zone paths. In those proofs, all the information about the tape of the Turing Machine (TM for short) is encoded into the timing information of the path, the sequence of pro-

	MTL	MITL	TCTL
Timed Automata	Undecidable [4]	EXPSPACE-c. [3]	PSPACE-c. [1]
Ult. Per. Zone Paths	Undecidable	EXPSPACE-c.	PSPACE-c.
Ult. Per. Region Paths	Undecidable	EXPSPACE-c.	PTIME
Finite Zone Paths	co-NP-c.	co-NP-c.	PSPACE-c.
Finite Region Paths	co-NP-c.	co-NP-c.	PTIME
Ult. Per. Timed Paths	PTIME	PTIME	PTIME
Finite Timed Paths	PTIME	PTIME	PTIME

Table 1. Complexity of path model checking

positional assignments being fixed by the ultimately periodic region path. This situation is rather different from the classical proofs in [4] and in [3]. Note also that the complexity of TCTL model checking goes from PSPACE-complete to PTIME in the case of region paths which is not the case for zone paths for which the problem stays PSPACE-complete. On the other hand, when we consider finite region or zone paths, the model checking problem for MTL becomes *decidable* and is co-NP-complete. The proofs for those results are based on (*i*) a *polynomial time* algorithm for checking the truth value of a MTL formula over a single finite timed path and on (*ii*) the proof that transitions between regions (respectively zones) in a region (respectively zone) path can be chosen nondeterministically in a finite subset of the rationals with *polynomially many* elements to establish if a region (respectively zone) path has at least one concretization that satisfies the MTL formula.

Related Work. Path model checking has been introduced in [10], where efficient algorithms are proposed for several untimed temporal logics (LTL, LTL+Past, ...). The basic remark there is that a CTL algorithm can be applied in order to verify LTL specifications, since path quantifier would refer to the only possible run of the structure. This does not hold here when we deal with region or zone paths.

Runtime verification and monitoring is another related issue. In that case, properties are verified on the fly during the run, as the events occur. Recently, monitoring algorithms have been proposed in the discrete-time framework for MTL [11].

In our work, we verify properties expressed in three important timed temporal logics. The case where the property is expressed as a timed automaton is treated in [5], where the authors show that deciding if a timed trace corresponds to an execution of a given timed automaton is PSPACE-complete. This result can easily be extended to region paths.

Structure of the Paper. The rest of the paper is structured as follows. In a first section, we define the classes of restricted sets of timed paths for which we study

the complexity of the model checking problems. We also recall in this section the syntax and semantics of MTL, MITL, and TCTL. In a second section, we present complexity results that can be interpreted as negative: we show that for some classes of restricted sets of timed models, some model checking problems are not easier than in the general case (when the set of timed paths is defined by a timed automaton). In a third section, we present complexity results that can be interpreted as positive: we show that for some interesting classes of restricted set of timed paths, some model checking problems are easier than in the general case.

1 Preliminaries

1.1 Timed Automata and Paths

We write \mathbb{R}^+ for the set of positive real numbers. In the sequel, all intervals we consider are convex subsets of \mathbb{R}^+ with rational greatest lower and least upper bounds. An interval is said to be *singular* if it contains only one value. Given two intervals I and J, and a positive rational number t, we write I - t for the set $\{x \in \mathbb{R}^+ \mid x + t \in I\}$ and I - J for the set $\{x \in \mathbb{R}^+ \mid \exists y \in J. x + y \in I\}$. Given an interval I, we note l(I) the greatest lower bound of I and r(I) least upper bound of I. An interval J follows an interval I if $I \cup J$ is convex, $I \cap J = \emptyset$ and r(I) = l(J). A finite (resp., infinite) sequence of intervals (I_i) , with $0 \le i \le n$ (resp. $0 \le i$) partitions a set $D \subseteq \mathbb{R}^+$ if for any $0 < i \le n$ (resp. 0 < i), interval $I_i = D$ (resp. $\bigcup_{i=0}^{i=+\infty} I_i = D$).

Let H be a set of variables. We define the set $\mathbb{C}(H)$ of clock difference constraints inductively as follows:

$$\mathbb{C}(H) \ni \delta, \delta' ::= x \sim c \mid x - y \sim c \mid \delta \land \delta'$$

for any two variables x and y in H, for ~ in $\{<, \leq, =, \geq, >\}$, and for any integer c.

Given a valuation v for the variables in H, the boolean value of a difference constraint δ is defined in the obvious way. Moreover, for any real t, we define the valuation v + t as being the valuation $x \mapsto v(x) + t$, and for any subset Cof H, the valuation v_C as being the valuation $x \in C \mapsto 0, x \notin C \mapsto v(x)$.

Definition 1. Given a set of states Q, and a set of clocks H, a timed path¹ $\tau = (q_i, v_i, I_i)$ is a (finite or infinite) sequence s.t.:

- $-(q_i)$ is a sequence of states in Q;
- (I_i) is a sequence of intervals forming a partition of \mathbb{R}^+ (or possibly of an interval [0, p] or [0; p) in the case of a finite path);
- $-v_i: H \to \mathbb{R}^+$ is the valuation of clocks in H when entering location q_i , at date $l(I_i)$. We require that, for each i and each clock x, either $v_{i+1}(x) = 0$ or $v_{i+1}(x) = v_i(x) + r(I_i) l(I_i)$;

¹ For the sake of brevity, we only consider dense time in the sequel. However, our results still hold when considering super-dense time [9].

- for each *i*, either $q_{i+1} \neq q_i$, or there exists a clock x s.t. $v_{i+1}(x) \neq v_i(x) + r(I_i) - l(I_i)$. This ensures that, at each step along that sequence, either we change location or we reset at least one variable².

A position along a timed path $\tau = (q_i, v_i, I_i)$ is a triple $(q, v, t) \in Q \times \mathbb{R}^H \times \mathbb{R}$ for which there exists an integer j s.t. $q = q_j$ and $v = v_j + t - l(I_j)$ and $t \in I_j$. For each $t \in \bigcup_i I_i$, there exists exactly one position (q, v, t) along τ , which we denote by $\tau(t)$. Given a timed path $\tau = (q_i, v_i, I_i)$ and a position (q_j, v, t) along ρ , the suffix of ρ starting at position (q_j, v, t) , denoted by $\rho^{\geq t}$, is the timed path (q'_i, v'_i, I'_i) where (1) $q'_i = q_{i+j}$ for all i, (2) $v'_i = v_{i+j}$ for i > 0, and $v'_0 = v$, (3) $I'_i = I_{i+j} - t$ for i > 0, and $I'_0 = ([t, +\infty) \cap I_j) - t$.

Definition 2. A timed automaton (*TA*) is a 6-tuple $\mathcal{A} = (Q, Q_0, H, l, \operatorname{Inv}, T, F)$ where: Q is a (finite) set of states; Q_0 is a subset of Q containing the set of initial states; H is a finite set of real-valued clocks; l is a function $Q \to 2^{AP}$ labeling each state with atomic propositions of AP; Inv is a function $Q \to \mathbb{C}(H)$ labeling each state with a set of timing constraints (called "invariants"); $T \subseteq Q \times \mathbb{C}(H) \times 2^H \times Q$ is a set of transitions; $F \subseteq Q$ is a subset of Q containing the set of accepting states.

In the sequel, we generally identify a location $q \in Q$ with its labeling l(q), if no ambiguity may arise from this notation. A *position* in a TA is a couple (q, v)where q is a state and v is a valuation of clocks in H satisfying Inv(q).

Definition 3. Given a set of states Q and a set of clocks H, a timed path (q_i, v_i, I_i) is a concretization of a TA $(Q, Q_0, H, l, \text{Inv}, T)$ if

- $-q_0 \in Q_0;$
- For each j, and for each $t \in I_j$, valuation $v_j + t l(I_j)$ satisfies $Inv(q_j)$;
- For each j, there exists a transition $(q_j, \varphi, C, q_{j+1}) \in E$ s.t. valuation $v_j + r(I_j) l(I_j)$ satisfies φ , and for all $x \in C$, $v_{j+1}(x) = 0$, and for all $x \in H \setminus C$, $v_{j+1}(x) = v_j(x) + r(I_j) l(I_j)$.
- either the timed path is infinite or its last state q_n is accepting, that is $q_n \in F$.

Definition 4. Two clock valuations v and v' are said to be equivalent w.r.t. a family $(c_x)_{x \in H}$ of constants, if the following conditions hold:

- for all clocks $x \in H$, either both v(x) and v'(x) are greater than c_x , or both have the same integer part;
- for all clocks $x \in H$, if $v(x) \leq c_x$, then $v(x) \in \mathbb{N}$ iff $v'(x) \in \mathbb{N}$;
- for all $x, y \in H$ with $v(x) \leq c_x$ and $v(y) \leq c_y$, if $\operatorname{fract}(v(x)) \leq \operatorname{fract}(v(y))$, then $\operatorname{fract}(v'(x)) \leq \operatorname{fract}(v'(y))$, where fract stands for the fractional part.

This obviously defines an equivalence relation. A *clock region* is an equivalence class for the equivalence relation between clocks. [2] proves that there are finitely many clock regions, more precisely at most $|H|! \cdot 4^{|H|} \cdot \prod_{x \in H} (c_x + 1)$.

² This conditions rules out "stuttering" paths. This is not restrictive as our logics, as you'll see later, cannot distinguish between timed traces with or without stuterring.

A clock region α is a *time-successor* of a clock region β if for each valuation $v \in \beta$, there exists a positive $t \in \mathbb{R}$ s.t. valuation v + t is in α , and for each t' s.t. $0 \le t' \le t$, valuation v + t' is in $\alpha \cup \beta$. It can be proved that, each clock region α has exactly one time-successor, which we will denote by succ(α) in the sequel. A clock region α is a *boundary class* if for any valuation $v \in \alpha$ and for any positive real t, valuation v + t is not in α .

Definition 5. Given a TA $\mathcal{A} = (Q, Q_0, H, l, \operatorname{Inv}, T, F)$, and the family (c_x) of maximal constants to which each clock x is compared in \mathcal{A} , the region graph $\mathcal{R}_{\mathcal{A}}$ of \mathcal{A} is the labeled graph (V, l', E) defined as follows:

- V is the product of the set of states of A and the set of clock regions;
- $-l': V \rightarrow 2^{AP}$ is defined by $l'(q, \alpha) = l(q);$
- *E* is the set of edges, containing two type of edges: Edges representing the elapse of time: for each vertex (q, α) in *V*, there is an edge to $(q, \operatorname{succ}(\alpha))$, if $\operatorname{succ}(\alpha)$ exists and contains a valuation satisfying the invariant $\operatorname{Inv}(q)$; Edges corresponding to transitions in \mathcal{A} : for each vertex (q, α) in *V*, for each edge (q, φ, C, q') in *T*, if there exists a valuation $v \in \alpha$ satisfying φ and s.t. v_C satisfies $\operatorname{Inv}(q')$, then there is an edge from (q, α) to (q', β) where β is the region containing valuation v_C .

Definition 6. A region path is a (finite or infinite) sequence $\rho = (q_i, \alpha_i)$ where q_i are locations and α_i are regions s.t. for all i either $\alpha_{i+1} = \text{succ}(\alpha_i)$, and $q_{i+1} = q_i$, or there exists a valuation $v \in \alpha_i$ and a set of clocks C s.t. $v_C \in \alpha_{i+1}$.

Definition 7. A zone is a convex union of regions. It can equivalently be defined as the set of clock valuations satisfying a difference constaint in $\mathbb{C}(H)$. A zone path is a (finite or infinite) sequence $\rho = (q_i, Z_i, C_i)$ where q_i are locations, Z_i are zones and C_i are the sets of clocks that are reset when entering Z_i .

A region (resp. zone) path π is said to be *ultimately periodic* (u.p. for short) if it can be written under the form $u \cdot v^{\omega}$, where u and v are finite region (resp. zone) paths. In both cases, finite paths are special cases of u.p. paths. A timed path is *ultimately periodic* if it is finite or if there exist two integers m and p > 0, and a real t, s.t. for any $i \ge m$, $q_{i+p} = q_i$, $v_{i+p} = v_i$, and $I_{i+p} = I_i + t$.

Note that a finite (or u.p.) region path is a special case of a TA, where states are pairs (q_i, α_i) , the set of initial states is the singleton $\{(q_0, \alpha_0)\}$, invariants are region constraints, clocks that are reset are clocks whose value is 0 when entering the target region, and the set of final states F is the last state pair (q_n, α_n) if the path is finite and is empty otherwise. A *concretization* of a region path is a concretization of the corresponding TA. The following proposition provides a simplified characterization.

Proposition 1. Let $\rho = (p_i, \alpha_i)_i$ be a region path. We say that a timed path $\pi = (q_j, v_j, I_j)_j$ is compatible with ρ , or is a concretization of ρ , iff (1) ρ and π are either both finite or both infinite, and for all k, $p_k = q_k$, (2) for all j, for all $t \in I_j$, valuation $v_j + t - l(I_j)$ belongs to region α_j .

Similarly, finite or u.p. zone paths form another subclass of the class of TA. We have the following simplified characterization of a *concretization* for a zone path:

Proposition 2. Let $\rho = (p_i, Z_i, C_i)_i$ be a zone path. We say that a timed path $\pi = (q_j, v_j, I_j)_j$ is compatible with ρ , or is a concretization of ρ , iff (1) ρ and π are either both finite or both infinite, and for all k, $p_k = q_k$, (2) for all k, for all $t \in I_k$, valuation $v_k + t - l(I_k)$ belongs to zone Z_k , (3) for all k, for all $x \in C_k$, $v_k(x) = 0$.

Note that a concretization of an u.p. region (or zone) path is generally not u.p. However, verifying that an u.p. timed path is a concretization of a region (or zone) path may be done in polynomial time [5].

1.2 Timed Temporal Logics

Definition 8. Let AP be a set of atomic propositions. The logic MTL is defined as follows:

$$\mathsf{MTL} \ni \varphi, \psi ::= \neg \varphi \mid \varphi \lor \psi \mid \varphi \mathbf{U}_I \psi \mid p \mid q \mid \dots$$

where I is an interval with integer greatest lower and least upper bounds and p,q,... belong to AP. The logic MITL is the sub-logic of MTL where intervals may not be singular.

MTL (and MITL) formulas are interpreted along timed paths³. Given a timed path $\tau = (q_i, v_i, I_i)$ and an MTL formula φ , we say that τ satisfies φ (written $\tau \models \varphi$) when:

$$\begin{array}{ll} \text{if } \varphi = p \ \text{then } p \in l(q_0) \\ \text{if } \varphi = \neg \xi \ \text{then } \tau \not\models \xi \\ \text{if } \varphi = \xi \lor \zeta \ \text{then } \tau \not\models \xi \text{ or } \tau \models \zeta \\ \text{if } \varphi = \xi \mathbf{U}_I \zeta \ \text{then } \text{there exists a position } (q, v, t) \text{ along } \tau \text{ s.t. } t \in I, \tau^{\geq t} \models \zeta \\ \text{and, for all } t' \in (0; t), \tau^{\geq t'} \models \xi. \end{array}$$

Standard unary modalities \mathbf{F}_I and \mathbf{G}_I are defined with the following semantics: $\mathbf{F}_I \xi \stackrel{\text{def}}{=} \top \mathbf{U}_I \xi$ and $\mathbf{G}_I \xi \stackrel{\text{def}}{=} \neg \mathbf{F}_I \neg \xi$, where \top is always true. We simply write \mathbf{F} and \mathbf{G} for $\mathbf{F}_{\mathbb{R}^+}$ and $\mathbf{G}_{\mathbb{R}^+}$, respectively.

Definition 9. Let \mathcal{A} be a TA, and φ be an MTL formula. The model checking problem defined by \mathcal{A} and φ consists in determining if, for any concretization τ of \mathcal{A} starting in an initial state, we have that $\tau \models \varphi$.

Definition 10. Let AP be a set of atomic propositions. The logic TCTL is defined as follows:

 $\mathsf{T}\mathsf{C}\mathsf{T}\mathsf{L}\ni\varphi,\psi\ ::=\ \varphi\vee\psi\mid\neg\varphi\mid\mathbf{E}\left(\varphi\,\mathbf{U}_{I}\,\psi\right)\mid\mathbf{A}\left(\varphi\,\mathbf{U}_{I}\,\psi\right)\mid p\mid q\mid...$

³ For the sake of simplicity, we interpret MTL (and MITL) formulas directly on timed paths instead of defining a notion of timed model where states and clocks are hidden.

where I is an interval with integer greatest lower and least upper bounds and p,q,... belong to AP.

TCTL formulas are interpreted at a position in a TA. Given a TA \mathcal{A} , a position (q, v) and a TCTL formula φ , we say that *position* (q, v) in \mathcal{A} satisfies φ , written $\mathcal{A}, (q, v) \models \varphi$, when:

$$\begin{array}{ll} \text{if } \varphi = p & \text{then } p \in l(q_0) \\ \text{if } \varphi = \neg \xi & \text{then } \mathcal{A}, (q, v) \not\models \xi \\ \text{if } \varphi = \xi \lor \zeta & \text{then } \mathcal{A}, (q, v) \models \xi \text{ or } \mathcal{A}, (q, v) \models \zeta \\ \text{if } \varphi = \mathbf{E} \left(\xi \mathbf{U}_I \zeta \right) & \text{then } \text{there exists a concretization } \tau = (q_i, v_i, I_i) \text{ of } \mathcal{A} \text{ s.t.} \\ q_0 = q & \text{and } v_0 = v, \text{ and a position } (q', v', t') \text{ along } \tau, \\ \text{s.t. } t' \in I, \mathcal{A}, (q', v') \models \zeta \text{ and all intermediate position} \\ \tau(t'') = (q'', v'', t'') \text{ with } 0 < t'' < t', \mathcal{A}, (q'', v'') \models \xi \\ \text{if } \varphi = \mathbf{A} \left(\xi \mathbf{U}_I \zeta \right) & \text{then } \text{ for any concretization } \tau = (q_i, v_i, I_i) \text{ of } \mathcal{A} \text{ with } q_0 = q \\ \text{ and } v_0 = v, \text{ there exists a position } (q', v', t') \text{ along } \tau, \\ \text{ s.t. } t' \in I, \mathcal{A}, (q', v') \models \zeta \text{ and all intermediate position} \\ \tau(t'') = (q'', v'', t'') \text{ with } 0 < t'' < t', \mathcal{A}, (q'', v'') \models \xi \\ \end{array}$$

We also define standard unary abbreviations $\mathbf{EF}_{I}\xi$, $\mathbf{AF}_{I}\xi$ and $\mathbf{EG}_{I}\xi$, $\mathbf{AG}_{I}\xi$ respectively as $\mathbf{E}(\top \mathbf{U}_{I}\xi)$, $\mathbf{E}(\top \mathbf{U}_{I}\xi)$ and $\neg \mathbf{AF}_{I}\neg\xi$, $\neg \mathbf{EF}_{I}\neg\xi$. We omit the subscript *I* when it equals \mathbb{R}^{+} .

Since region and zone paths can be seen as TA, satisfaction of a TCTL formula at a position along a region or zone path is defined in the obvious way. Note that contrary to the untimed case [10], TCTL is not equivalent to MTL along a region or zone path, since such a path contains (infinitely) many timed paths.

Definition 11. Let \mathcal{A} be a TA, (q, v) be a position of \mathcal{A} , and φ be a TCTL formula. The model-checking problem defined by \mathcal{A} , (q, v) and φ consists in determining if \mathcal{A} , $(q, v) \models \varphi$.

In the sequel, for the two problems defined above, we consider the subcases where \mathcal{A} is (*i*) a single finite (or u.p.) timed path, (*ii*) a finite (or u.p.) region path, (*iii*) a finite (or u.p.) zone path.

2 Negative Results

The main goal of restricting to subclasses of TA is to obtain feasible algorithms for problems that are hard in the general case. This section presents cases where our restrictions are not sufficient and do not reduce complexity.

2.1 Linear Time Logics Along Ultimately Periodic Region Paths

What we expected most was that model checking MTL would become decidable along an u.p. region path. This is not the case, as shown in Theorem 1. The proof



Fig. 1. Encoding of the tape of a Turing Machine

of this theorem requires an encoding of a TM computation by timing information only. Remember that the proof for the general model checking problem (for sets of models defined by TA) is simply a reduction from the satisfiability problem of MTL. The technique needed here is different: We encode the tape of an unbounded TM on a unit-length path by an atomic proposition being true for a strictly positive (but as small as we want) amount of time. MTL can distinguish between those two cases, and allows us to ensure that the path really encodes a computation of the TM. See Fig. 1 for an example.

Theorem 1. Model checking a MTL formula along an u.p. region path is undecidable.

Proof. This is done by encoding the acceptance problem for a TM (does \mathcal{M} accept w?) to the problem of verifying a MTL formula along a region path. Wlog, we assume that the alphabet has only two letters $\{a, b\}$, and a special symbol # for empty cells. Since the ordering of atomic propositions along the path is fixed, the contents of the tape has to be encoded through timing informations only. Since we have no bound on the total length needed for the computation, encoding of one letter must be arbitrarily compressible. Encoding of an a is done by atomic proposition q being true at only one precise moment (with duration 0), while b is encoded by q being true for a positive amount of time. An atomic proposition p is used in the same way for indicating the beginning and end of the encoding of the tape. See top of Fig. 1 for an example. For any atomic proposition x, we write $x^+ = x \mathbf{U}_{>0} \top$ and $x^0 = x \land \neg x^+$. Then a is encoded with p^+ and b with p^0 .

A third letter, r, is used for encoding the position of the control head: r^+ is true (between p and q) at the position where the control head stands, and r^+ is false everywhere else. Encoding the control state (s_k , for some k between 0 and n-1) is done through n 1-time-unit-long slices of the path. Along each slice, q^+ and r^+ will never be satisfied; p^+ will be true only in the k+1-th slice, meaning that the current control state is s_k , and false everywhere else. Fig. 1 shows a complete encoding of one configuration. The configuration separator will be the only slice where d^+ will hold, for a fourth atomic proposition d. There is one last



Fig. 2. The region path ρ

atomic proposition, b, used for filling up all the gaps. The region path generating such an encoding is shown on Fig. 2.

With this encoding, it is possible to write MTL formulas ensuring the correct behavior of the TM. $\hfill \Box$

In the same way, MITL model checking problems are not easier with u.p. region paths than in the general case. Again, the proof for the general model checking problem is a reduction from the satisfiability problem for MITL. Here, we cannot proceed that way and must encode the computation of an exponential space TM using a single region path and an MITL formula.

Theorem 2. Model checking an MITL formula along an u.p. region path is EXPSPACE-complete.

2.2 TCTL Along Finite or Ultimately Periodic Zone Paths

Since zones are more general than regions, hardness results for region paths extend to zone paths. Thus model checking MITL and MTL along a zone path is respectively EXPSPACE-complete and undecidable.

Regarding TCTL, the algorithm we propose for region paths (see Section 3.3) could be extended to zone paths, but would result in an exponential explosion in the number of states (since a zone may contain an exponential number of regions). In fact, this explosion cannot be avoided (unless PTIME=PSPACE), since we have the following result:

Theorem 3. Model checking TCTL along an ultimately periodic zone path is *PSPACE-complete*.

3 Positive Results

Restricting to paths sometimes allows for more efficient algorithms. This happens for MTL and MITL along single timed paths as well as along finite region or zone paths, and for TCTL along u.p. region paths.

3.1 Linear Time Logics and Timed Paths

Along a timed path, all quantitative information is precisely known, and model checking MTL can be performed quite efficiently.

Theorem 4. Model checking MTL along a u.p. timed path is in PTIME.

Proof. Consider a finite⁴ timed path $\tau = (q_i, v_i, I_i)_{i=0..p}$. The idea is to compute, for each subformula ψ of the MTL formula φ under study, the set of reals t s.t. $\tau^{\geq t} \models \psi$. We represent this set S_{ψ} as a union (which we prove is finite) of intervals whose interiors are disjoint.

The sets $S_{\psi} = \{J_i^{\psi}\}$ are computed recursively as follows:

- For atomic propositions, the intervals are trivially computed by "reading" the input path;
- For boolean combinations of subformulas, they are obtained by applying the corresponding set operations, and then possibly merging some of them in order to get disjoint intervals. Obviously the union of two families $\bigcup_{i=1}^{m} I_i$ and $\bigcup_{i=1}^{n} J_j$ of intervals contains at most m+n intervals, and the complement of $\bigcup_{i=1}^{m} I_i$ contains at most m+1 intervals. Thus the intersection of $\bigcup_{i=1}^{m} I_i$ and $\bigcup_{j=1}^{n} J_j$ contains at most m+n+3 intervals;
- and $\bigcup_{j=1}^{n} J_j$ contains at most m + n + 3 intervals. Thus the intersection of $\bigcup_{i=1}^{n} I_i$ and $\bigcup_{j=1}^{n} J_j$ contains at most m + n + 3 intervals; - For subformulas of the form $\varphi \mathbf{U}_I \psi$, the idea is to consider, for each interval $J_i^{\varphi} \in S_{\varphi}$ and each interval $J_j^{\psi} \in S_{\psi}$, the interval $((\overline{J_i^{\varphi}} \cap J_j^{\psi}) - I) \cap J_i^{\varphi}$. It precisely contains all points in J_i^{φ} satisfying $\varphi \mathbf{U}_I \psi$ with a witness for ψ in J_i^{ψ} .

This construction seems to create $|S_{\varphi}| \cdot |S_{\psi}|$ intervals, but a more careful enumeration shows that it only creates at most $|S_{\varphi}| + |S_{\psi}| + 3$: indeed, the procedure only creates at most one interval for each *non-empty* interval $\overline{J_i^{\varphi}} \cap J_j^{\psi}$, and the intersection of $\bigcup_{i=1}^m \overline{I_i}$ and $\bigcup_{j=1}^n J_j$ contains at most m+n+3 intervals.

At the end of this procedure, S_{φ} contains $O(p \cdot |\varphi|)$ intervals, and $\tau \models \varphi$ iff 0 is in one of these intervals. Our algorithm thus runs in time $O(|\tau| \cdot |\varphi|)$. \Box

Timed paths could be seen as timed automata if rational difference constraints were allowed in guards and invariants. In that case, the semantics of TCTL along a timed path would have been equivalent to the semantics of MTL, since timed automaton representing a timed path would be completely deterministic.

3.2 MTL and MITL Along Finite Region and Zone Paths

The difficulty for model checking MTL along infinite u.p. region or zone paths was that we had to remember precise timing information about the (infinite, not periodic) concretization against which we verify the MTL formula. In the finite case, we prove we only have to guess and remember a finite (in fact, polynomial) amount of information, making the problem decidable:

Lemma 1. Model checking MTL along a finite zone path is in co-NP.

⁴ We describe our algorithm only for finite paths, but it can easily be extended to infinite u.p. paths, by reasoning symbolically about the periodic part.

Proof. We prove that the *existential* model checking problem is in NP, which is equivalent. The basic idea is to non-deterministically guess the dates t_i at which each of the *n* transitions is fired. Once these dates are known, we have a timed path and we can check in polynomial time that this path is a concretization of the initial zone path and that it satisfies the MTL formula (see Theorem 4).

What remains to be proved is that t_i 's can be chosen in polynomial time, *i.e.* the number of non-deterministic steps is polynomial. To that purpose, we consider an MTL formula φ , and prove that if φ is true along the region path, *i.e.* if there exist timestamps *s.t.* the corresponding timed path satisfies φ , then there exists timestamps in the set $\{p/(n+1) \mid 0 \leq p \leq (n+1) \cdot (c_Z + c_{\varphi})\}$ where *n* is the number of states in the zone path, c_Z is the sum of the constants appearing in the zone path and c_{φ} is the sum of the constants appearing in φ .

The proof of this last statement is as follows: the set of (in)equalities t_i 's must satisfy are: (In)equalities related to the zone path: when t_i 's are "fixed", we can compute all valuations of clocks along the zone path. The constraints those valuations must satisfy give constraints that t_i 's must satisfy. These constraints have the form $a \le t_i - t_j \le b$ or $a \le t_i \le b$; (In)equalities related to the formula: for each subformula, we can compute a set of *disjoint* time intervals (depending on t_i 's) in which the subformula is true (see proof of Theorem 4).

This leads to a disjunction of difference constraints, which has a solution iff the formula is true along one concretization of the finite zone path. Since a difference constraints cannot distinguish between two equivalent valuations (for the equivalence of Definition 4), if there exists a solution, any equivalent valuation of t_i 's is a solution. This ensures that if there is a solution, then there is a solution in $\{p/(n+1) \mid p \in \mathbb{N}\}$. Moreover, each date can be bounded with the sum of all the constants appearing in the zone path or in the formula: Indeed, constraints between t_i 's only involves constants lower than this sum. Thus the dates can be guessed in polynomial time.

This algorithm is in fact optimal, and we have the following result:

Theorem 5. Model checking MTL or MITL along finite region (or zone) paths is co-NP-complete.

The co-NP-hardness proof is similar to the one of Theorem 3, and consists in encoding 3-SAT into an (existential) model checking problem.

3.3 TCTL Along Ultimately Periodic Region Paths

We prove that TCTL properties can be verified in polynomial time along region paths. This contrasts with the negative results we got previously for MTL and MITL, and intuitively relies on the fact that, contrary to MTL, we don't have to "remember" the precise values of the clocks when we fire a transition, since path quantifiers are applied to all modalities of the formula.

In this section, we describe our algorithm. It first requires to compute temporal relations between any two regions. **Definition 12.** Let $\rho = (\rho)_i$ be a region path. Given two integers k and l, we say that a real d is a possible delay between regions ρ_k and ρ_l if there exists a concretization $\pi = (p_j, v_j, I_j)_j$ of ρ , and a real t, s.t. $t \in I_k$ and $t + d \in I_l$. We write delay (ρ, k, l) for the set of possible delays between ρ_k and ρ_l along ρ .

The following two lemmas prove that possible delays form an interval with integer bounds:

Lemma 2. Given a region path ρ and two integers k and l, delay (ρ, k, l) is an interval.

Lemma 3 ([7]). Let ρ be a region path, k, l and c be three integers. If there exists $d \in (c; c+1)$ s.t. $d \in \text{delay}(\rho, k, l)$, then $(c, c+1) \subseteq \text{delay}(\rho, k, l)$.

There remains to compute both upper and lower bounds. [8] designed algorithms for computing minimum and maximum delays between valuations and regions. We could apply them in our case. However, their algorithms would compute delays between regions of a finite structure, and we need to compute delays between any two regions of the infinite, u.p. path.

It happens that possible delays in an u.p. region path are u.p., but won't necessarily have the same initial and periodic parts. Below, we compute a table containing the minimum and maximum delays between one region and any future region, by computing those delays for a finite set of regions until a periodicity is detected. Thus, we build a table containing "initial" delays of the minimal and maximal paths, plus the length and duration of their periodic parts.

Lemma 4. Let $\rho = u \cdot v^{\omega}$ be an u.p. region path. We can effectively build in time $O(|u|^2 \cdot |H|)$ the table containing all the necessary information for computing $(i, j) \mapsto \text{delay}(\rho, i, j)$.

Proof. We build the region graph G of the product of ρ , seen as a timed automaton, and T_1 shown on Fig. 3. Graph G is *not* u.p. in the general case: see Fig. 4 for an example.

Since we add one new clock which is bounded by 1, the total number of regions is at most multiplied by 2(1 + |H|), corresponding to the 2(1 + |H|) possible ways of inserting fract(t) among the fractional parts of the other clocks.

In automaton T_1 , t is the fractional part of the total time elapsed since the beginning of the path, and the number of times t has been reset is the integral part of that total time. Extracting the minimal and maximal delay paths is now an easy task, since in each region of G:

- either fract(t) = 0, and possibly two transitions may be firable: one corresponding to letting time elapse, going to a region where t > 0, and the other one corresponding to the transition in ρ ;



Fig. 3. Automaton T_1



(b) The resulting region automaton (highlighted states are states appearing in the minimal or maximal delay path)

	$ ho_0$	ρ_1	ρ_2	ρ_3	ρ_4	$ ho_5$	ρ_2'	ρ'_3	$ ho_4'$	$ ho_5'$	period
min. delay	0	0+	0+	0+	0+	0+	0+	0+	0+	0+	0 t.u., 4 states
max. delay	0	1-	1-	1-	1-	2-	2-	2-	2-	3-	1 t.u., 4 states

(c) Delays from the initial region

Fig. 4. Computation of possible delays between regions

- or fract(t) > 0, and clock t can't reach value 1 in that region, because another clock will reach an integer value before; The only possible outgoing edge is the transition of the original region path;
- or fract(t) > 0, and clock t can reach value 1 (and then be reset to 0). Two cases may arise: resetting t might be the only outgoing transition, or there could be another possible transition derived from the original region path. If there are two outgoing edges, firing the transition that resets t amounts

to letting time elapse, and firing the other transition amounts to running as quickly as possible.

In all cases, we also have the condition that we cannot cross two successive immediate transitions, since the resulting region path would not have any concretization.

Now, the maximal delay path is obtained by considering the path where we always select the transition corresponding to time elapsing, *i.e.* resetting t or switching from t = 0 to 0 < t < 1, when such a transition is available; The minimal delay path is the one we get when always selecting the other transition. Moreover, those minimal and maximal delay paths are u.p., since G has finitely many regions and the paths are built deterministically. They have at most $|u| + 2(|H| + 1) \cdot |v|$ regions in their initial part and at most $2(|H| + 1) \cdot |v|$ regions in their periodic part.

From these paths, we can build a table containing all relevant information for computing minimal and maximal delays between the initial region and any region along ρ (see Fig. 4(c)). Any value inbetween is a possible delay thanks to lemma 2. Computing this table takes time $O(|u| + 2(|H| + 1) \cdot |v|)$. Computing possible delays between any two states along ρ can be achieved by repeating the above procedure starting from the first |u| + |v| states of ρ (since removing longer prefixes gives rise to the same paths), thus in total time $O((|u| + 2(|H| + 1) \cdot |v|) \cdot (|u| + |v|)) \subseteq O(|H| \cdot |\rho|^2)$.

Theorem 6. Model checking a TCTL formula φ along an u.p. region path ρ can be done in polynomial time (more precisely $O(|\varphi| \cdot |\rho| \cdot |H| + |H| \cdot |\rho|^2)$).

Proof. This is achieved by a labeling algorithm. We label region ρ_i of ρ with subformula ψ of φ iff $\rho_i \models \psi$. This is not ambiguous as a TCTL formula cannot distinguish between two equivalent valuations [1].

The labeling procedure runs in time $O(|\varphi| \cdot |\rho| \cdot |H|)$. Since delays between regions must be computed, the global TCTL model checking problem along u.p. region paths can be performed in time $O(|\varphi| \cdot |\rho| \cdot |H| + |H| \cdot |\rho|^2)$.

References

- R. Alur, C. Courcoubetis, and D. L. Dill. Model-Checking in Dense Real-Time. Information and Computation, 104(1), pages 2–34, Academic Press, May 1993.
- [2] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2), pages 183–235, Elsevier Science, Apr. 1994.
- [3] R. Alur, T. Feder, and Th. A. Henzinger. The Benefits of Relaxing Punctuality. *Journal of the ACM*, 43(1), pages 116–146, ACM Press, Jan. 1996.
- [4] R. Alur and Th. A. Henzinger. A Really Temporal Logic. *Journal of the ACM*, 41(1), pages 181–203, ACM Press, Jan. 1994.
- [5] R. Alur, R. P. Kurshan, and M. Viswanathan. Membership Question for Timed and Hybrid Automata. In *Proc. 19th Symp. Real-Time Systems (RTS'98)*, Dec. 1998, pages 254–263. IEEE Comp. Soc. Press, Dec. 1998.

- [6] A. Bouajjani, S. Tripakis, and S. Yovine. On-the-Fly Symbolic Model Checking for Real-Time Systems. In Proc. 18th Symp. Real-Time Systems (RTS'97), Dec. 1997, pages 25–35. IEEE Comp. Soc. Press, Dec. 1997.
- [7] V. Bruyère, E. Dall'Olio, and J.-F. Raskin. Durations, Parametric Model Checking in Timed Automata with Presburger Arithmetic. In H. Alt and M. Habib, eds, *Proc. 20th Symp. Theoretical Aspects of Computer Science (STACS 2003)*, Feb.– Mar. 2003, vol. 2607 of *LNCS*, pages 687–698. Springer Verlag, Feb. 2003.
- [8] C. Courcoubetis and M. Yannakakis. Minimum and Maximum Delay Problems in Real-Time Systems. *Formal Methods in System Design*, 1(4), pages 385–415, Kluwer Academic, Dec. 1992.
- [9] Z. Manna and A. Pnueli. Verifying Hybrid Systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, eds, *Hybrid Systems*, vol. 736 of *LNCS*, pages 4–35. Springer Verlag, 1993.
- [10] N. Markey and Ph. Schnoebelen. Model Checking a Path (Preliminary Report). In R. Amadio and D. Lugiez, eds, *Proc. 14th Intl Conf. Concurrency Theory* (*CONCUR 2003*), Aug.-Sept. 2003, vol. 2761 of *LNCS*, pages 251–265. Springer Verlag, Aug. 2003.
- [11] P. Thati and G. Roşu. Monitoring Algorithms for Metric Temporal Logic Specifications. In K. Havelund and G. Roşu, eds, *Proc. 4th Intl Workshop on Runtime Verification (RV 2004)*, Apr. 2004, ENTCS, pages 131–147. Elsevier Science, Apr. 2004.

Asynchronous Games 2: The True Concurrency of Innocence

Paul-André Melliès

Equipe Preuves Programmes Systèmes CNRS & Université Paris 7

Abstract. In game semantics, one expresses the higher-order value passing mechanisms of the λ -calculus as sequences of atomic actions exchanged by a Player and its Opponent in the course of time. This is reminiscent of trace semantics in concurrency theory, in which a process is identified to the sequences of requests it generates. We take as working hypothesis that game semantics is, indeed, the trace semantics of the λ -calculus. This brings us to a notion of asynchronous game, inspired by Mazurkiewicz traces, which generalizes the usual notion of arena game. We then extract the true concurrency semantics of λ -terms from their interleaving semantics formulated as innocent strategies. This reveals that innocent strategies are positional strategies regulated by forward and backward interactive confluence properties. We conclude by defining a non uniform variant of the λ -calculus, whose game semantics is formulated as a trace semantics.

1 Introduction

Game semantics has taught us the art of converting the *higher-order* value passing mechanisms of the λ -calculus into sequences of *atomic* interactions exchanged by a Player and its Opponent in the course of time. This metamorphosis of higher-order syntax into interactive semantics has significantly sharpened our understanding of the simply-typed λ -calculus, either as a pure calculus, or as a calculus extended with programming features like recursion, conditional branching, local control, local states, references, non determinism, probabilistic choice, etc.

Game semantics is similar to *trace semantics* in concurrency theory. A process is commonly described as a symbolic device which interacts with its environment by emitting or receiving requests. A sequence of such requests is called a *trace*. The trace semantics of a process π is defined as the set of traces generated by this process. In many cases, this semantics characterizes the contextual behaviour of the process.

Game semantics develops quite the same story for the λ -calculus. The terminology changes obviously: requests are called *moves*, and traces are called *plays*. But everything works as in trace semantics: the semantics of a λ -term M of type A is the set of plays σ generated by the λ -term M; and this set σ characterizes

the contextual behaviour of the λ -term. One original aspect of game semantics however, not present in trace semantics, is that the type A defines a game, and that the set σ defines a *strategy* of that game.

The starting point of this work is that game semantics is *really* the trace semantics of the λ -calculus. The thesis is apparently ingenuous. But it is surprisingly subversive because it prescribes to reevaluate a large part of the technical and conceptual choices accepted in game semantics... in order to bridge the gap with concurrency theory. Three issues are raised here:

- 1. The treatment of duplication in mainstream game semantics (eg. in arena games) distorts the bond with trace semantics, by adding justification pointers to traces. According to our methodology, this particular treatment of duplication should be revisited. This is done in the first article of our series on asynchronous games [21]. We recall below the indexed and group-theoretic reformulation of arena games operated there.
- 2. Thirty years ago, a theory of *asynchronous traces* was formulated by Antoni Mazurkiewicz in order to relate the *interleaving* and *true concurrency* semantics of concurrent computations. Game semantics delivers an interleaving semantics of the λ -calculus, formulated as innocent strategies. What is the corresponding true concurrency semantics? The task of this second article on asynchronous games is to answer this question precisely.
- 3. Ten years ago, a series of full abstraction theorems for PCF were obtained by characterizing the interactive behaviour of λ -terms as either innocent, or history-free strategies, see [3, 13, 24]. We feel that the present work is another stage in the "full abstraction" program initiated by Robin Milner [23]. For the first time indeed, we do not simply characterize, but also derive the syntax of λ -terms from elementary causality principles, expressed in asynchronous transition systems. This reconstruction requires the mediation of [21] and of its indexed treatment of threads. This leads us to an *indexed* and *non-uniform* λ -calculus, from which the usual λ -calculus follows by group-theoretic principles. In this variant of the λ -calculus, the game semantics of a λ -term may be directly formulated as a trace semantics, performing the syntactic exploration or parsing of the λ -term.

The Treatment of Duplication. The language of traces is limited, but sufficient to interpret the *affine* fragment of the λ -calculus, in which every variable occurs at most once in a λ -term. In this fragment, every trace (=play) generated by a λ -term is an alternating sequence of received requests (=Opponent moves) and emitted requests (=Player moves). And a request appears at most once in a trace.

The extension from the affine fragment to the whole λ -calculus requires to handle semantically the duplication mechanisms. This is a delicate matter. Several solutions have been considered, and coexist today in the litterature. By way of illustration, take the λ -term chosen by Church to interpret the natural number 2:

$$M = \lambda f.\lambda x.ffx$$

In front of two λ -terms *P* and *Q*, the λ -term *M* duplicates its first argument *P*, and applies it twice to its second argument *Q*. This is performed syntactically by two β -reductions:

$$MPQ \longrightarrow_{\beta} (\lambda x.PPx)Q \longrightarrow_{\beta} PPQ$$
 (1)

Obviously, the remainder of the computation depends on the λ -terms P and Q. The game-theoretic interpretation of the λ -term M has to anticipate all cases. This requires to manipulate several threads of the λ -term P simultaneously — and many more than two copies when the λ -term $P_{(1)}$ uses its first argument $P_{(2)}$ several times in $P_{(1)}P_{(2)}Q$.

Now, the difficulty is that each thread of P should be clearly distinguished. A compact and elegant solution has been introduced by Martin Hyland, Luke Ong and Hanno Nickau, in their *arena games* [13, 24]. We recall that an *arena* is a forest, whose nodes are the *moves* of the game, and whose branches $m \vdash n$ are oriented in order to express the idea that the move m justifies the move n. A move n is *initial* when it is a root of the forest, or alternatively, when there is no move m such that $m \vdash n$. A justified play is then defined as a pair $(m_1 \cdots m_k, \varphi)$ consisting of a sequence of moves $m_1 \cdots m_k$ and a partial function $\varphi : \{1, ..., k\} \rightarrow \{1, ..., k\}$ providing the so-called *pointer* structure. The partial function φ associates to every occurrence i of a non-initial move m_i the occurrence $\varphi(i)$ of a move $m_{\varphi(i)}$ such that $m_{\varphi(i)} \vdash m_i$. One requires that $\varphi(i) < i$ to ensure that the justifying move $m_{\varphi(i)}$ occurs before the justified move m_i . Finally, the partial function φ is never defined on the occurrence i of any initial move m_i .

The pointer structure φ provides the necessary information to distinguish the several threads of a λ -term in the course of interaction — typically the several threads or copies of P in example (1). The pointer structure φ is conveniently represented by drawing "backward pointers" between occurrences of the sequence $m_1 \cdots m_k$. By way of illustration, consider the arena $m \vdash n \vdash p$ in which the only initial move is m. A typical justified play (s, φ) of this arena is represented graphically as:

$$m \cdot n \cdot p \cdot n \cdot p \cdot n \cdot p + m \cdot n \cdot p \qquad (2)$$

Because adding justification pointers distorts the bond with trace semantics, in particular with Mazurkiewicz traces, we shift in [21] to another management principle based on *thread indexing*, already considered in [3, 12]. The idea is to assign to each copy of the λ -term *P* in example (1) a natural number $k \in \mathbb{N}$ (its index) which characterizes the thread among the other copies of *P*. In the case of the justified play (2), this amounts to (a) adding a dumb move \bigstar in order to justify the initial moves of the sequence, (b) indexing every justification pointer of the resulting sequence with a natural number:



then finally (c) encoding the sequence (3) as the sequence of indexed moves below:

$$m_{17} \cdot n_{17,5} \cdot p_{17,5,69} \cdot n_{17,4} \cdot p_{17,4,20} \cdot n_{17,1} \cdot p_{17,5,7} \cdot m_5 \cdot n_{5,70} \cdot p_{17,4,4}. \tag{4}$$

Obviously, the translation of a justified play (s, φ) depends on the choice of indices put on its justification pointers. Had we not taken sides with trace semantics and concurrency theory, we would be tempted (as most people do in fact) to retract to the notation (2) which is arguably simpler than its translation (4). But we carry on instead, and regulate the indexing by asking that two justification pointers starting from different occurrences *i* and *j* of the same move *n*, and ending on the same occurrence $\varphi(i) = \varphi(j)$, receive different indices *k* and *k'*. This indexing policy ensures that every indexed move occurs at most once in the sequence (4). In this way, we are back to the simplicity of the affine fragment of the λ -calculus.

An interesting point remains to be understood: what can be said about two different encodings of the same justified play? The first article of our series [21] clarifies this point. Every game is equipped with a left and right group actions on moves:

$$G \times M \times H \longrightarrow M \qquad (g, m, h) \mapsto g \cdot m \cdot h$$
(5)

where *M* denotes the set of (indexed) moves, and *G* and *H* the two groups acting on *M*. Intuitively, the right (resp. left) group action operates on an indexed move $m_{k_0,...,k_j}$ by altering the indices k_{2i} assigned by Opponent (resp. the indices k_{2i+1} assigned by Player). The *orbit* of an (indexed) move $m_{k_0,...,k_j}$, is precisely the set of all (indexed) moves of the form $m_{k'_0,...,k'_j}$. Now, the action of $g \in G$ and $h \in H$ on (indexed) moves induces a left and right action on plays, defined pointwise:

$$g \cdot (m_1 \cdots m_k) \cdot h = (g \cdot m_1 \cdot h) \cdots (g \cdot m_k \cdot h)$$
(6)

It appears that the justified plays of the original arena game coincide with the orbits of plays modulo the left and right group actions. Typically, the justified play (2) is just the play (4) modulo pointwise group action (6). One nice contribution of this second article on asynchronous games, is to explain the *syntactic* meaning of the group action (5). This is done in a non-uniform variant of the λ -calculus introduced in Section 6.

Asynchronous Traces. After these necessary preliminaries on thread indexing, we shift to the core of this article: true concurrency vs. interleaving in game semantics. Two requests a and b are called *independent* in a process π when they can be emitted or received by π in any order, without interference. Independence

of *a* and *b* is represented graphically by *tiling* the two sequences $a \cdot b$ and $b \cdot a$ in the 2-dimensional diagram:



The *true concurrency* semantics of a process π is then extracted from its *interleaving* semantics, by quotienting the traces of π modulo the *homotopy equivalence* ~ obtained by permuting independent requests. Expressing concurrency by permuting events is a pervading idea in concurrency theory. It originates from the work of Antoni Mazurkiewicz on asynchronous traces over a partially ordered alphabet [18, 19] and appears in the theory of asynchronous transition systems [25, 15, 27] as well as in rewriting theory [20]. The *n*-dimensional presentation of the idea, and the connection to (directed) homotopy in cubical sets, is formulated in [26, 10].

In comparison, mainstream game semantics is still very much 1-dimensional. By way of illustration, take the sequential boolean game \mathbb{B} , starting by an Opponent question q followed by a Player answer true or false:



The plays of the tensor product $\mathbb{B} \otimes \mathbb{B}$ are obtained by interleaving the plays of the two instances \mathbb{B}_1 and \mathbb{B}_2 of \mathbb{B} . Thus, (a fragment of) the game $\mathbb{B} \otimes \mathbb{B}$ looks like this:



We point out in [22] that the two plays in (9) are different from a *procedural* point of view, but equivalent from an *extensional* point of view, since both of them realize the "extensional value" (*true, false*). We thus bend the two paths, and tile the resulting 2-dimensional octagon as follows:



By doing so, we shift from usual sequential games played on trees, to sequential games played on *directed acyclic graphs* (dags). This enables us to analyze the extensional content of sequential games, and to obtain a game-theoretic proof of Ehrhard's collapse theorem [9].

However instructive, the framework developed in [22] is not entirely satisfactory, because the permutation tiles are *global* — that is, they involve more than two moves in general. In contrast, the asynchronous game model presented here admits only *local* permutations tiles, similar to tile (7). By way of illustration, this decomposes the global tile (10) into four local tiles:



It is interesting that by shifting from (10) to (11), concurrent plays like $q_1 \cdot q_2$ appear in the model. From our point of view, this means that a satisfactory theory of sequentiality requires a concurrent background.

The Non-uniform λ -Calculus. Here comes the most surprising, most difficult, and maybe most controversial, part of the paper. In Section 2, we define an asynchronous game as an event structure whose events are polarized +1 for Player moves and -1 for Opponent moves. This polarization of events gives rise to a new class of events $m \cdot n$ consisting of an Opponent move m followed by a Player move n. We call *OP-moves* any such pair of moves. Just like ordinary moves, two *OP*-moves $m_1 \cdot n_1$ and $m_2 \cdot n_2$ may be permuted in a play, in the following way:



The permutation diagram (12) induces an homotopy relation \sim_{OP} between plays. The dual relation \sim_{PO} is defined symmetrically, by permuting *PO*-moves $n \cdot m$ instead of *OP*-moves, where by *PO*-move $n \cdot m$ we mean an Opponent move n followed by a Player move m. Note that both \sim_{OP} and \sim_{PO} preserve *alternation* of plays.

Now, there is a well-established theory of *stable* asynchronous transition systems, see for instance [25, 15, 20], in which every sequence of transitions s is characterized (modulo homotopy) as a directed acyclic graph of so-called *canonical forms*. The canonical form of a transition a in a sequence $s \cdot a$ of transitions, expresses the cascade of transitions necessary for the enabling of the transition a. Formally, a sequence of transitions $t \cdot a$ is a canonical form of $s \cdot a$ when (1)
$s \cdot a \sim t \cdot a \cdot t'$ for some t', and (2) whenever $t \sim t' \cdot b$, then a cannot be permuted before b. The stability property ensures that this canonical form $s \cdot a$ is *unique*.

The theory may be applied to the asynchronous transition system with *OP*-moves as transitions, which happens to be stable. From this follows that every *OP*-move $m \cdot n$ in an alternating play $s \cdot m \cdot n$ has a unique *canonical form*. Strikingly, this canonical form is precisely the so-called *Player view* $\lceil s \cdot m \cdot n \rceil$ of the play $s \cdot m \cdot n$, introduced by Martin Hyland *et al.* in arena games [13, 24] and adapted to asynchronous games in Section 3.

We claim that here lies the essence of the syntax of the λ -calculus. It has been already noted in [7] that every Player view of a justified play (s, ϕ) corresponds to the branch of an η -long Böhm tree. When adapted to the indexed treatment of threads described in [21] and recalled above, the correspondence defines the branch of a *non-uniform* η -long Böhm tree. The definition of the non-uniform λ -calculus is given in Section 6. A nice feature of the calculus is that the strategy σ associated to a non-uniform λ -term may be defined in the same way as a trace semantics. This is also done in Section 6.

Related Works. The idea of relating a dynamic and a static semantics of interaction is formulated for the first time by Patrick Baillot *et al.* in [6]. The idea reappears implicitly in the concurrent game semantics introduced by Samson Abramsky and the author [5], in which games are complete lattices of positions, and strategies are closure operators. As a closure operator, every strategy is at the same time an increasing function on positions (the dynamic point of view) and a set of positions (the static point of view). The present paper is the result of a long journey (five years!) to connect this concurrent game semantics to mainstream sequential game semantics. See also [2].

Martin Hyland and Andrea Schalk develop in [14] a notion of games on graphs quite similar to the constructions presented here and in [22]. One difference is the treatment of duplication: backtracking in [14, 22], repetitive and indexed here. From this choice follows that the permutation tilings are global in [14, 22] whereas they are local here. Another difference is that our positions are defined as *ideals* of moves.

Outline. In the remainder of the article, we define our notion of asynchronous game (Section 2) and adapt the usual definition of innocent strategy to our setting (Section 3). We then characterize the innocent strategies in two ways: diagrammatically (Section 4) and positionally (Section 5). This leads to a non-uniform variant of the λ -calculus, for which we define a trace semantics, and which we relate to the usual λ -calculus (Section 6). Finally, we deliver a series of refinements of asynchronous games (Section 7).

2 Asynchronous Games

We choose the simplest possible definition of *asynchronous game*, in which the only relation between moves is an order relation \leq which reformulates the *justi*-

fication structure of arena games. This is enough to describe the language PCF, a simply-typed λ -calculus enriched with arithmetic, conditional branching, and recursion. Other more expressive variants are discussed in section 7.

Event Structures. An event structure is an ordered set (M, \leq) such that every element $m \in M$ defines a finite downward-closed subset $m \downarrow = \{n \in M \mid n \leq m\}$.

Asynchronous Games. An asynchronous game is a triple $A = (M_A, \leq_A, \lambda_A)$ consisting of:

- an event structure (M_A, \leq_A) whose elements are called the *moves* of the game,
- a function $\lambda_A : M_A \longrightarrow \{-1, +1\}$ which associates to every move a *polarity* +1 (for the Player moves) or -1 (for the Opponent moves).

Positions. A *position* of an asynchronous game A is any *finite* downward closed subset of (M_A, \leq_A) .

The Positional Lattice. The set of positions of A is denoted $\mathcal{D}(A)$. Since positions are ordered by inclusion, and closed under finite union, the partial order $(\mathcal{D}(A), \subseteq)$ defines a sup-lattice. The empty position, which is the least element of $(\mathcal{D}(A), \subseteq)$, is denoted $*_A$. Positions are also closed under arbitrary *nonempty* intersection. Adding a top element \top to $(\mathcal{D}(A), \subseteq)$ provides a neutral element to intersection, and induces a *complete* lattice $\mathcal{D}(A)^{\top} = (\mathcal{D}(A), \subseteq)^{\top}$. The greatest least bound and least upper bound of a family $(x_i)_{i \in I}$ of positions in $\mathcal{D}(A)$ are computed respectively as:

$$\bigwedge_{i \in I} x_i = \begin{cases} \top & \text{if } I \text{ is empty,} \\ \bigcap_{i \in I} x_i \text{ otherwise,} \end{cases}$$
$$\bigvee_{i \in I} x_i = \begin{cases} \top & \text{if } \bigcup_{i \in I} x_i \text{ is infinite,} \\ \bigcup_{i \in I} x_i \text{ if } \bigcup_{i \in I} x_i \text{ is finite.} \end{cases}$$

We call $\mathcal{D}(A)^{\mathsf{T}}$ the *positional lattice* associated to the game A.

The Positional Graph. Every asynchronous game A induces a graph $\mathcal{G}(A)$:

- whose nodes are the positions $x, y \in \mathcal{D}(A)$,
- whose edges $m : x \longrightarrow y$ are the moves verifying $y = x + \{m\}$, where + denotes disjoint union, or equivalently, that $y = x \cup \{m\}$ and that the move *m* is not element of *x*.

We call this graph $\mathfrak{G}(A)$ the *positional graph* of the game A. We write $s : x \rightarrow y$ for a path

 $x \xrightarrow{m_1} x_1 \xrightarrow{m_2} \cdots \xrightarrow{m_{k-1}} x_{k-1} \xrightarrow{m_k} y$

between two positions x and y. Note that there is no repetition of move in the sequence:

$$\forall i, j \in \{1, ..., k\}, \qquad i \neq j \Rightarrow m_i \neq m_j.$$

The target y of the path $s: x \rightarrow y$ may be deduced from the source x and the sequence of moves $m_1, ..., m_k$, using the equation:

$$y = x + \bigcup_{1 \leq i \leq k} \{m_i\}.$$

A path of $\mathcal{G}(A)$ is thus characterized by its source (or alternatively, its target) and the sequence of moves $m_1 \cdots m_k$.

Homotopy. Given two paths $s, s' : x \rightarrow y$ in $\mathcal{G}(A)$, we write $s \sim^1 s'$ when $s = m \cdot n$ and $s' = n \cdot m$ for two moves $m, n \in M_A$. The *homotopy equivalence* ~ between paths is defined as the least equivalence relation containing \sim^1 , and closed under composition; that is, for every four paths $s_1 : x_1 \rightarrow x_2$ and $s, s' : x_2 \rightarrow x_3$ and $s_2 : x_3 \rightarrow x_4$:

$$s \sim s' \implies s_1 \cdot s \cdot s_2 \sim s_1 \cdot s' \cdot s_2.$$

We also use the notation \sim in our diagrams to indicate that two (necessarily different) moves m and n are permuted:

 $\begin{array}{c}
 n \\
 y_1 \\
 m \\
 x \\
 n
 x
 n
 (13)$

Note that our current definition of asynchronous game implies that two paths $s_1 : x_1 \rightarrow y_1$ and $s : x_2 \rightarrow y_2$ are homotopic iff $x_1 = x_2$ and $y_1 = y_2$. Thus, homotopy becomes informative only in the presence of an *independence* relation between moves, see Section 7.

Alternating Paths. A path $m_1 \cdots m_k : x \rightarrow y$ is alternating when:

 $\forall i \in \{1, \dots, k-1\}, \quad \lambda_A(m_{i+1}) = -\lambda_A(m_i).$

Alternating Homotopy. Given two paths $s, s': x \rightarrow y$ in $\mathcal{G}(A)$, we write $s \sim_{OP}^{1} s'$ when $s = m_1 \cdot n_1 \cdot m_2 \cdot n_2$ and $s' = m_2 \cdot n_2 \cdot m_1 \cdot n_1$ where the moves $m_1, m_2 \in M_A$ are Opponent and the moves moves $n_1, n_2 \in M_A$ are Player. The situation is summarized in diagram (12). The relation \sim_{OP} is defined as the least equivalence relation containing \sim_{OP}^{1} and closed under composition. Note that $s \sim_{PO} s'$ implies $s \sim s'$, but that the converse is not true, since in diagram (12) one has $m_1 \cdot n_1 \cdot m_2 \cdot n_2 \sim m_1 \cdot n_2 \cdot m_2 \cdot n_1$ without having $m_1 \cdot n_1 \cdot m_2 \cdot n_2 \sim_{OP} m_1 \cdot n_2 \cdot m_2 \cdot n_1$. *Plays.* A play is a path starting from the empty position $*_A$:

$$*_A \xrightarrow{m_1} x_1 \xrightarrow{m_2} \cdots \xrightarrow{m_{k-1}} x_{k-1} \xrightarrow{m_k} x_k$$

in the positional graph $\mathcal{G}(A)$. The set of plays is noted P_A .

Equivalently, a play of A is a finite sequence $s = m_1 \cdots m_k$ of moves, without repetition, such that the set $\{m_1, ..., m_j\}$ is downward closed in (M_A, \leq_A) for every $1 \leq j \leq k$.

Strategy. A strategy σ is a set of alternating plays of even length such that:

- the strategy $s \in \sigma$ contains the empty play,
- every nonempty play $s \in \sigma$ starts by an Opponent move,

 $-\sigma$ is closed by even-length prefix:

 $\forall s \in P_A, \forall m, n \in M_A, \quad s \cdot m \cdot n \in \sigma \Rightarrow s \in \sigma,$

 $-\sigma$ is deterministic: $\forall s \in P_A, \forall m, n, n' \in M_A$,

$$s \cdot m \cdot n \in \sigma \text{ and } s \cdot m \cdot n' \in \sigma \quad \Rightarrow \quad n = n'.$$

We write σ : A when σ is a strategy of A.

3 Innocent Strategies

The notion of *innocence* has been introduced by Martin Hyland, Luke Ong and Hanno Nickau in the framework of arena games [13, 24]. It is designed to capture the interactive behaviour of the simply-typed λ -calculus with a constant Ω for non-termination, either formulated as η -long Böhm trees [7], as proofs of Polarized Linear Logic [17], or (after a continuation-passing style translation) as PCF programs augmented with local control [16, 4, 11]. Asynchronous games enable to reformulate the notion of *innocence* in a concurrency friendly way. The original definition of innocence is based on the notion of *Player view* of a justified play (s, φ) , defined using the pointer structure φ . In asynchronous games, the situation is slightly simpler than in arena games, because the play sis non repetitive. In particular, there is no need to distinguish a move m from its occurrences in the play. More: every play s comes with an implicit pointer structure φ derived from the causality relation \leq between moves, as follows.

Justification Pointers. Suppose that m and n are two different moves of an asynchronous game A. We write $m \vdash_A n$, and say that m justifies n, when:

 $-m \leq_A n$, and

- for every move $p \in M_A$ such that $m \leq_A p \leq_A n$, either m = p or p = n.

View Extraction. We define the binary relation $\stackrel{\text{OP}}{\rightsquigarrow}$ as the smallest relation between alternating plays such that:

$$s_1 \cdot m \cdot n \cdot s_2 \stackrel{\text{OP}}{\leadsto} s_1 \cdot s_2$$

for every alternating play s_1 and nonempty path s_2 such that m is an Opponent move which does not justify any move in s_2 , and n is a Player move which does not justify any move in s_2 .

Player View. The relation $\stackrel{\text{OP}}{\leadsto}$ defines a noetherian and locally confluent rewriting system on alternating plays. By Newman's lemma, the rewriting system is confluent. Thus, every alternating play $s \in P_A$ induces a unique normal form noted $\lceil s \rceil \in P_A$ and called its *Player view:*

$$s \stackrel{\text{OP}}{\leadsto} s_1 \stackrel{\text{OP}}{\leadsto} \cdots \stackrel{\text{OP}}{\leadsto} s_k \stackrel{\text{OP}}{\leadsto} \ulcorner s \urcorner$$

Asynchronous Innocence. A strategy σ is innocent in an asynchronous game A when for every plays $s, t \in \sigma$, for every Opponent move $m \in M_A$ and Player move $n \in M_A$:

$$s \cdot m \cdot n \in \sigma$$
 and $t \cdot m \in P_A$ and $\lceil s \cdot m \rceil \sim_{OP} \lceil t \cdot m \rceil \Rightarrow t \cdot m \cdot n \in \sigma$.

Asynchronous innocence is equivalent to usual innocence in the intuitionistic fragment [13, 24]. In that fragment, indeed, every move has at most one justifying move, and thus, the two Player views $\lceil s \cdot m \rceil$ and $\lceil t \cdot m \rceil$ are \sim_{OP} -equivalent iff they are equal. On the other hand, asynchronous innocence generalizes the usual notion of innocence to more "concurrent" arenas, in which several moves $n_1, ..., n_k$ may justify the same move m — a situation which does not occur in arena games associated to intuitionistic types.

4 Diagrammatic Innocence

In this section, we reformulate diagrammatically the notion of *innocence* in asynchronous games.

Backward Consistency. A strategy σ is called *backward consistent* when for every play $s_1 \in P_A$, for every path s_2 , for every moves $m_1, n_1, m_2, n_2 \in M_A$, it follows from

$$s_1 \cdot m_1 \cdot n_1 \cdot m_2 \cdot n_2 \cdot s_2 \in \sigma$$
 and $\neg (n_1 \vdash_A m_2)$ and $\neg (m_1 \vdash_A m_2)$

that

$$\neg(n_1 \vdash_A n_2)$$
 and $\neg(m_1 \vdash_A n_2)$ and $s_1 \cdot m_2 \cdot n_2 \cdot m_1 \cdot n_1 \cdot s_2 \in \sigma$.

TEAM LING

Forward Consistency. A strategy σ is called *forward consistent* when for every play $s_1 \in P_A$ and for every moves $m_1, n_1, m_2, n_2 \in M_A$, it follows from

 $s_1 \cdot m_1 \cdot n_1 \in \sigma$ and $s_1 \cdot m_2 \cdot n_2 \in \sigma$ and $m_1 \neq m_2$

that

$$n_1 \neq n_2$$
 and $s_1 \cdot m_1 \cdot n_1 \cdot m_2 \cdot n_2 \in \sigma$.

We prove by a diagrammatic reasoning inspired by Rewriting Theory that, for every strategy σ of an asynchronous game A:

Proposition 1 (Diagrammatic Characterization). The strategy σ is innocent iff it is backward and forward consistent.

5 **Positional Innocence**

We establish below the main result of the paper: innocent strategies are positional strategies (Theorem 2). We then characterize innocent strategies as positional strategies (Proposition 3) and identify them as concurrent strategies in the sense of [5] (Proposition 4).

Positional Strategy. A strategy σ : A is called *positional* when for every two plays $s_1, s_2 : *_A \rightarrow x$ in the strategy σ , and every path $t : x \rightarrow y$ of $\mathcal{G}(A)$, one has:

 $s_1 \sim s_2$ and $s_1 \cdot t \in \sigma \implies s_2 \cdot t \in \sigma$.

Every positional strategy is characterized by the set of positions of $\mathcal{D}(A)$ it reaches, defined as:

$$\sigma^{\bullet} = \{ x \in \mathcal{D}(A), \exists s \in \sigma, s : *_A \twoheadrightarrow x \}.$$

Theorem 2 (Positionality). Every innocent strategy σ is positional.

The positional characterization of innocence (Proposition 3) works in any asynchronous game in which justification is alternated, that is, where $m \vdash n$ implies $\lambda(n) = -\lambda(m)$ for every move m and n. In particular, it works in any interpretation of a formula of intuitionistic linear logic.

Proposition 3 (Positional Characterization). A positional strategy σ is innocent iff the set σ^{\bullet} of positions satisfies:

 $-\sigma^{\bullet}$ is closed under intersection: $x, y \in \sigma^{\bullet} \Rightarrow x \cap y \in \sigma^{\bullet}$,

- $\begin{array}{l} -\sigma^{\bullet} \text{ is closed under union: } x, y \in \sigma^{\bullet} \Rightarrow x \cup y \in \sigma^{\bullet}, \\ -\text{ forward confluence: if } \sigma^{\bullet} \ni x \xrightarrow{m} w \Rightarrow z \in \sigma^{\bullet} \text{ and } m \text{ is an Opponent move,} \\ \text{ then there exists a unique Player move } w \xrightarrow{n} y \text{ such that } \sigma^{\bullet} \ni y \Rightarrow z \in \sigma^{\bullet}, \end{array}$
- backward confluence: if $\sigma^{\bullet} \ni x \twoheadrightarrow w \xrightarrow{n} z \in \sigma^{\bullet}$ and n is a Player move, then there exists a unique Opponent move $y \xrightarrow{m} w$ such that $\sigma^{\bullet} \ni x \twoheadrightarrow y \in \sigma^{\bullet}$,
- initial condition: $*_{\mathbf{A}}$ is element of σ^{\bullet} .

Proposition 4. Every innocent strategy $\sigma : A$ defines a closure operator σ^{\bullet} on the complete lattice $\mathcal{D}(A)^{\top}$ of positions.

This series of properties explicates the true concurrency nature of innocence. Proposition 4 bridges sequential arena games with concurrent games as formulated in [5]. In particular, positionality implies that strategies may be composed just as relations, or as cliques in the hypercoherence space model [8].

If the reader finds the idea of *positionality* difficult to grasp, we hope that the Proposition below will clarify the situation. It is quite straightforward to define a notion of *innocent* counter-strategy τ interacting against the strategy σ . The counter-strategy τ may *withdraw* at any stage of the interaction. Every such withdraw of τ induces an even-length play $s : *_A \to x$ in the strategy τ , whose target position $x \in \tau^{\bullet}$ is of even cardinality. Our next result states that the static evaluation (by intersection) of σ^{\bullet} against τ^{\bullet} coincides with the dynamic evaluation (by interaction) of σ against τ .

Proposition 5. For every position $x \in \mathcal{D}(A)$:

$$\sigma^{\bullet} \cap \tau^{\bullet} = \{x\} \iff \sigma \cap \tau = \{s\} and s : *_A \twoheadrightarrow x.$$

It is nearly routine to construct a category \mathcal{G} with asynchronous games as objects, and innocent strategies as morphisms. The only difficulty is to interpret the exponentials, which is done by equipping every game with a left and right group action, in the spirit of [21]. The resulting category \mathcal{G} defines a model of intuitionistic linear logic without additives. The usual category of arena games and innocent strategies [13, 24] embeds fully and faithfully (as a cartesian closed category) in the kleisli category associated to the category \mathcal{G} and to its comonad !.

6 The Non-uniform λ -Calculus

We introduce a non-uniform variant of the λ -calculus. It is called *non-uniform* because the argument of a function $\lambda x.P$ is not a λ -term Q, but a vector \overrightarrow{Q} of λ -terms Q_i where $i \in \mathbb{N}$ is an index for each occurrence x(i) (or function call) of the variable x in P. The calculus is affine in nature (never two occurrences of x(i) occur in the same term), but the simply-typed λ -calculus may be encoded in it, thanks to group-theoretic ideas developed in our first article on asynchronous games [21].

Definition of the Calculus. The non-uniform λ -terms *P* and vectors of arguments \vec{Q} are defined by mutual induction:

$$\begin{array}{c|c} P ::= x(i) & \text{located variable} \\ & | & P \overrightarrow{Q} & \text{application} \\ & | & \lambda x.P & \text{abstraction} \end{array}$$

 $\vec{Q} ::= (Q_i)_{i \in \mathbb{N}}$ vector of non-uniform λ -terms indexed by an integer $i \in \mathbb{N}$

where a located variable x(i) consists of a variable x in the usual sense, and an integer $i \in \mathbb{N}$. We require that every located variable x(i) appears at most once in a term. Note that a non-uniform λ -term is generally infinite. The β -reduction is defined as

$$(\lambda x.P) \overrightarrow{Q} \longrightarrow_{\beta} P[x(i) := Q_i]$$

where $P[x(i) := Q_i]$ denotes the non-uniform λ -term obtained by replacing each located variable x(i) in P by the non-uniform λ -term Q_i . The non-uniform λ -terms are typed by the simple types of the λ -calculus, built on the base type α :

$$x(i): A \vdash x(i): A \qquad \frac{\Gamma \vdash P: A \Rightarrow B \qquad (\Delta_i \vdash Q_i: A)_{i \in \mathbb{N}}}{\Gamma, \Delta_0, \Delta_1, \Delta_2, \dots \vdash P \overrightarrow{Q}: B}$$
$$\frac{\Gamma, x(i_0): A, x(i_1): A, x(i_2): A, \dots \vdash P: B}{\Gamma \vdash \lambda x. P: A \Rightarrow B}$$

Here, a context $\Gamma, \Delta, ...$ may contain an infinite number of located variables, since the \Rightarrow -elimination rule involves a family of derivation trees $(\Delta_i \vdash Q_i : A)_{i \in \mathbb{N}}$. The point is that the \Rightarrow -introduction rule may migrate an infinite number of located variables x(i) from the context to the λ -term.

Non-Uniform η *-Long Böhm Trees.* The non-uniform η *-long Böhm trees of simple type* $A = A_1 \Rightarrow \cdots A_m \Rightarrow \alpha$ *are of three kinds:*

- 1. $\lambda x_1 \dots \lambda x_m$. $(y(i) \overrightarrow{Q_1} \cdots \overrightarrow{Q_n})$ where
 - every variable x_j is of type A_j for $1 \le j \le m$,
 - the located variable y(i) is of type $B = B_1 \Rightarrow \cdots B_n \Rightarrow \alpha$ for some type B,
 - every non uniform η -long Böhm tree $(Q_k)_i$ is of type B_k , for $1 \le k \le n$ and $i \in \mathbb{N}$.
- 2. or Ω_B where Ω_B is a fixed constant of type *B*,
- 3. or $\lambda x_1 \dots \lambda x_m$. U where U is a fixed constant of type α , and every variable x_j is of type A_j , for $1 \le j \le m$.

Trace Semantics. We describe a trace semantics for non-uniform η -long Böhm trees, which coincides with the game semantics delivered by our asynchronous game model. The Opponent moves are generated by the rule

$$R^- : \Omega_A \longrightarrow \lambda x_1 \cdots \lambda x_m.$$
 U

where $A = A_1 \Rightarrow \cdots A_m \Rightarrow \alpha$ and the variable x_j is of type A_j for every index $1 \le j \le m$. The Player moves are generated by the rule

$$R^+_{x(i)} : \ \mho \longrightarrow \ x(i) \ \overrightarrow{\varOmega}_{A_1} \cdots \overrightarrow{\varOmega}_{A_m}$$

where x(i) is a located variable of type $A = A_1 \Rightarrow \cdots A_m \Rightarrow \alpha$, and $\overrightarrow{\Omega}_{A_j}$ is the vector which associates to every index $i \in \mathbb{N}$ the constant Ω_{A_j} , for every

 $1 \le j \le m$. Last point, every move from an η -long Böhm tree is labelled by a subtree of the type *A*, once translated in linear logic as an infinite formula, using the equation $A \Rightarrow B = !A \multimap B$, and the definition of the exponential modality as infinite tensor: $!A = \bigotimes_{i \in \mathbb{N}} A$.

Uniformity and Bi-invariance. The usual (uniform) η -long Böhm trees of the λ -calculus are extracted from their non-uniform counterpart by applying a *bi-invariance* principle introduced in [21]. As recalled in the introduction, see (5), every game there is equipped with a left and right group action on moves. A strategy σ is called *bi-invariant* when, for every play $s \in \sigma$ and every right action $h \in H$, there exists a left action $g \in G$ such that $g \cdot s \cdot h \in \sigma$. This characterizes the strategies which are "blind to thread indexing", and thus the strategies which behave as if they were defined directly in an arena game. The concept of bi-invariance remains formal and enigmatic in [21]. Here, quite fortunately, the non-uniform λ -calculus provides a simple syntactical explanation to this concept of bi-invariance, what we explain now.

Every intuitionistic type A defines a left and right group action (5) on the asynchronous game [A] interpreting it in the asynchronous game model. These two group actions may be understood syntactically as acting on the non-uniform η -long Böhm trees P of type A, as follows: the effect of a right group action $h \in H$ is to permute the indices inside the vectors of arguments \vec{Q} in P, while the effect of a left group action $q \in G$ is to permute the indices of the located variables x(i) in P. By analogy with [21], a non-uniform η -long Böhm tree P is called *bi-invariant* when for every permutation $h \in H$, there is a permutation $g \in G$ such that $q \cdot P \cdot h = P$. It is not difficult to see that an η -long Böhm tree in the usual λ -calculus is just a *bi-invariant* η -long Böhm tree in the non-uniform λ -calculus, modulo left group action (that is, permutation of the indices of the located variables.) For instance, let P_j denote the non-uniform η -long Böhm tree $P_j = \lambda x \cdot \lambda y \cdot (x(j) \overrightarrow{y})$ of type $A = (\alpha \Rightarrow \alpha) \Rightarrow (\alpha \Rightarrow \alpha)$, where \overrightarrow{y} associates to every index $i \in \mathbb{N}$ the located variable y(i). Obviously, P_i is bi-invariant, and represents the uniform η -long Böhm tree $\lambda x \cdot \lambda y \cdot x y$ of same type A. Note that P_i is equivalent to any P_k modulo left group action. The trace (or game) semantics of P_i is given by:

$$\Omega_A \xrightarrow{m} \lambda x.\lambda y. \mho \xrightarrow{n} \lambda x.\lambda y. (x(j) \overrightarrow{\Omega}_{\alpha}) \xrightarrow{m_k} \lambda x.\lambda y. (x(j) \overrightarrow{Q_k}) \xrightarrow{n_k} \cdots$$

Here, the move *m* by Opponent (labelled by the type *A*) asks for the value of the head variable of P_j , and the move *n* by Player (labelled by the type $(\alpha \Rightarrow \alpha)_j$) answers x(j); then, the move m_k by Opponent (labelled by α_k in $(\alpha \Rightarrow \alpha)_j$) asks for the value of the head variable of the *k*-th argument of x(j), inducing the vector of arguments $(Q_k)_i = \Omega_\alpha$ for $i \neq k$ and $(Q_k)_k = \mho$; finally the move n_k by Player (labelled by α_k) answers y(k), etc... This example illustrates the fact that the trace (or game) semantics of a non-uniform η -long Böhm tree is simply the exploration (or parsing) of that tree by the Opponent.

7 Additional Structures

For clarity's sake, we deliver the simplest possible definition of asynchronous game in Section 2. We review below possible extensions of this definition.

Compatibility. One may add an *incompatibility* relation # between moves, in order to obtain a model of intuitionistic linear logic with additives. The relation # indicates when two moves cannot appear in the same position, and thus cannot appear in the same play. The coherence axiom $(m_1 \# m_2 \le m_3 \implies m_1 \# m_3)$ is required for every moves m_1, m_2, m_3 , just as in event structures [27].

Independence. There is a well-established tradition in trace semantics to describe *interference* mechanisms using an independence relation I between events [19]. Similarly, an independence relation between moves may be added to asynchronous games, in order to study interference in imperative programming languages. Take the game model of Idealized Algol presented in [1]. Suppose that an independence relation indicates that the moves read and write (n) are interfering in the interpretation of the variable type var, for every natural number n. In that case, the interference between read and write (n) induces obstructions ("holes") to the homotopy relation ~ on the game var. Quite interestingly, the asynchronous definition of innocence adapts smoothly, and remains compositional in the presence of interfering moves (that is, it defines a category).

8 Conclusion

The theory of asynchronous games is designed to bridge the gap between mainstream game semantics and concurrency theory. Our preliminary results are extremely encouraging. We establish indeed that the cardinal notion of sequential game semantics: *innocence*, follows from elementary principles of concurrency theory, formulated in asynchronous transition systems. We deduce from this a non-uniform λ -calculus, whose game semantics is expressed as a trace semantics. This provides a concurrency-friendly picture of the λ -calculus, and new diagrammatic foundations for the understanding of its syntax and semantics.

References

- [1] S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions, 1997.
- [2] Samson Abramsky. Sequentiality vs. concurrency in games and logic. Report Research Report RR-01-08, Oxford University, Programming Research Group, April 2001.
- [3] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Information and Computation*, 163(2):409–470, 2000.
- [4] Samson Abramsky and Guy McCusker. Game Semantics. Springer Verlag, 1999.

- [5] Samson Abramsky and Paul-André Melliès. Concurrent games and full completeness. In *Logic in Computer Science 99*, pages 431–442, Trento, July 1999. IEEE Computer Society Press.
- [6] Patrick Baillot, Vincent Danos, Thomas Ehrhard, and Laurent Regnier. Timeless games. In Morgen Nielsen and Wolfgang Thomas, editors, *Proceedings of CSL'97*, number 1414 in Lecture Notes in Computer Science, pages 56–77, Aarhus, 1997. Springer Verlag.
- [7] Vincent Danos, Hugo Herbelin, and Laurent Regnier. Games semantics and abstract machines. In *Proceedings of the* 11th Symposium on Logic in Computer Science, pages 394–405, New Brunswick, 1996. IEEE Computer Society Press.
- [8] Thomas Ehrhard. Hypercoherences: a strongly stable model of linear logic. *Mathematical Structures in Computer Science*, 3(4):365–385, 1993.
- [9] Thomas Ehrhard. A relative definability result for strongly stable functions and some corollaries. *Information and Computation*, 1997.
- [10] Eric Goubault. Geometry and concurrency: A user's guide. *Mathematical Structures in Computer Science*, 10(4), August 2000.
- [11] R. Harmer. Games and full abstraction for nondeterministic languages. Phd thesis, University of London, 2000.
- [12] Martin Hyland. *Game Semantics*. Publications of the Newton Institute. Cambridge University Press, 1997.
- [13] Martin Hyland and Luke Ong. On full abstraction for PCF: I, II and III. Information and Computation, 163(2):285–408, December 2000.
- [14] Martin Hyland and Andrea Schalk. Games on graphs and sequentially realizable functionals. In *Logic in Computer Science 02*, pages 257–264, Kopenhavn, July 2002. IEEE Computer Society Press.
- [15] Dietrich Kuske. Non deterministic automata with concurrency relations and domains. In Proceedings of the Colloquium on Trees in Algebra and Programming, CAAP'94, volume 787 of Lecture Notes in Computer Science. Springer Verlag, 1994.
- [16] James Laird. Full abstraction for functional languages with control. In *Logic in Computer Science*, pages 58–67, 1997.
- [17] Olivier Laurent. Polarized games (extended abstract). In Proceedings of the seventeenth annual symposium on Logic In Computer Science, pages 265–274, Copenhagen, July 2002. IEEE Computer Society Press.
- [18] Antoni Mazurkiewicz. Concurrent program schemes and their interpretations. Technical Report DAIMI PB 78, Aarhus University, 1977.
- [19] Antoni Mazurkiewicz. *The book of traces*, chapter Introduction to trace theory. World Scientific Publishing, 1995.
- [20] Paul-André Melliès. Axiomatic rewriting 4: a stability theorem in rewriting theory. In *Logic in Computer Science '98*. IEEE Computer Society Press, July 1998.
- [21] Paul-André Melliès. Asynchronous games 1: a group-theoretic formulation of uniformity. Prépublication électronique PPS//04/06//n°31 (pp), Equipe Preuves, Programmes et Systèmes, April 2003.
- [22] Paul-André Melliès. Sequential algorithms and strongly stable functions. Prépublication électronique PPS//03/09//n°23 (pp), Equipe Preuves, Programmes et Systèmes, April 2003. To appear in the special issue "Game Theory Meets Theoretical Computer Science" of *Theoretical Computer Science*.
- [23] Robin Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

- [24] Hanno Nickau. Hereditarily sequential functionals. In A. Nerode and Yu. V. Matiyasevich, editors, *Proceedings of the Symposium on Logical Foundations of Computer Science: Logic at St. Petersburg*, volume 813 of *Lecture Notes in Computer Science*, pages 253–264. Springer Verlag, 1994.
- [25] E. W. Stark P. Panangaden, V. Shanbhogue. Stability and sequentiality in data flow networks. In A. Nerode and Yu. V. Matiyasevich, editors, *International Conference on Automates, Languages and Programming,* volume 443 of *Lecture Notes in Computer Science,* pages 253–264. Springer Verlag, 1990.
- [26] Vaughn Pratt. Modeling concurrency with geometry. In *Proceedings of the eighteenth annual symposium on Principles Of Programming Languages*, pages 311– 322. ACM, IEEE Computer Society Press, January 1991.
- [27] G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, volume* 4. Oxford University Press, 1995.

Open Maps, Alternating Simulations and Control Synthesis

Paulo Tabuada

Department of Electrical Engineering University of Notre Dame Notre Dame, IN 46556 ptabuada@nd.edu

Abstract. Control synthesis is slowly transcending its traditional application domain within engineering to find interesting and useful applications in computer science. Synthesis of interfaces, distributed network monitors or reactive programs are some examples that benefit from this design paradigm. In this paper we shed new light on the interplay between the fundamental notion of bisimulation and the control synthesis problem. We first revisit the notion of alternating simulation introduced by Alur and co-workers as it naturally captures important ingredients of the control synthesis problem. We then show that existence of controllers enforcing specifications through bisimulation, alternating simulation or simulation can be characterized by the existence of certain alternating simulations and bisimulations between the specification and the system to be controlled. These results highlight and unify the role of simulations and bisimulations in the control synthesis setting for a wide range of concurrency models. This is achieved by developing our study within the framework of open maps. We illustrate our results on transition systems and timed transition systems.

1 Introduction

Computer Science and Control Theory. The control synthesis problem is the central theme of control theory. The traditional setup consists of a system, usually modeled by a differential equation with certain inputs that can be freely assigned, and a specification. The objective is to synthesize a controller, which based on the observation of the current system state, changes the system inputs in order to alter its behavior and to enforce the specification. However, many man made systems are not adequately described by differential equations and in the late 80's Ramadage and Wonham initiated the application of control theoretic ideas to the control of systems described by finite state automata [1]. Even though a different model is used, the same control synthesis problem was shown to be relevant in this context. As introduced by Ramadge and Wonham, the control synthesis problem consists in synthesizing a supervisor finite state automaton P, modeling the system to be controlled, recognizes a specified regular language S.

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 466-480, 2004.

[©] Springer-Verlag Berlin Heidelberg 2004

If one interprets P, S and C as software models, the same problem immediately suggests different applications within computer science such as synthesis of interfaces between software modules [2], distributed monitoring of networks [3], synthesis of reactive embedded controllers [4], etc.

Approximately at the same time that Ramadage and Wonham were obtaining the first results on supervisory control, a similar problem was being investigated in the computer science community: Pnueli and Rosner considered synthesis of reactive software [5,6]. Synthesis of software from (temporal logic) specifications had already been addressed by the computer science community [7,8] for closed systems. Independently of the (computer or control) perspective, it is the author's belief that control synthesis problems benefit from the different approaches and contributions originating from computer science and control communities.

Motivation. In this paper we revisit the control synthesis problem in a branching time framework with 3 main objectives:

- Unify control synthesis results across several different concurrency models such as transition systems, asynchronous transition systems, probabilistic transition systems, timed transition systems, Petri nets, etc.
- Highlight the fundamental role played by the notions of bisimulation, alternating simulation and simulation in control synthesis problems.
- Reduce decidability and complexity of control synthesis to decidability and complexity of bisimulation, alternating simulation and simulation.

To accomplish the first objective, we develop our results within the general framework of open maps introduced by Joyal and co-workers [9]. Open maps provide a unified language to discuss and prove results for a large class of apparently different concurrency models. We will use transition systems as a source of motivation and examples throughout the paper and we will also apply our results to timed transition systems which underlie timed automata. However, the general framework of open maps allows to export the presented results to other classes of concurrency models as described in [10, 9, 11, 12].

The second objective motivated us to generalize Alur and co-workers [13] notion of alternating simulation to the open maps framework. Such generalization provides the right language to formulate the control synthesis problem by considering the environment as an opponent trying to violate the specification. The proposed notion coincides with Alur and co-workers notion for transition systems and provides notions of alternating simulation for other classes of concurrency models through the co-reflections introduced in [10]. Such notions and corresponding logic characterizations remain largely unexplored as we focus, in this paper, on the control synthesis problem.

The open maps framework was also crucial in highlighting the similarities and differences between the different versions of the control synthesis problem we have considered. We studied three natural requirements to be enforced by control: bisimulation, alternating simulation and simulation. For each different requirement, we show that existence of a controller is characterized by existence of a bisimulation, alternating simulation or simulation between the specification and the system to be controlled. In addition to unifying existing results and to highlight the role of bisimulation and similar notions, the developed results also allow to reduce decidability and complexity of control synthesis to decidability and complexity of bisimulation and related notions.

Related Work. The control synthesis problem for transition systems in a branching time framework has been shown to be decidable by Madhusudan and Thiagarajan in [14]. The main ingredient was the characterization of controllers in terms of good subgraphs and strong subgraphs whose existence can be decided. However, it was not clear in [14] how such objects depend on the underlying concurrency model (transition systems) neither how they relate with alternating simulations. Our results show that such graphs correspond in fact to certain simulations and bisimulations between specification and the system to be controlled. Furthermore, by reformulating existence results in terms of such well known notions, the results become applicable to other classes of systems where these notions make sense. The relation between bisimulation and supervisory control problems was also discussed in [15]. However, bisimulation was used as a way to efficiently compute controllers in a linear time framework, rather than as an essential ingredient for branching time. A different approach was discussed in [16] using co-algebraic methods. Even though bisimulation was used in a fundamental way, through co-inductive definitions and proofs, the approach is rather different from the one considered in this paper. In [16], the adversarial effect of disturbances is captured by a new composition operator rather than by the use of alternating simulations. It is therefore not possible to understand how the requirements for the existence of controllers can be weakened by weakening the required relation between specification and controlled system. Supervisory controllers in branching time were also considered in [17], however failure semantics was used instead of bisimulation to specify the desired behavior. Other lines of research in branching time scenarios considered supervisory control problems for CTL or CTL* specifications [18-20].

2 The Model

The control synthesis problem can naturally be viewed as a game between the *controller* and the *environement*. To provide motivation for the abstract setup used throughout the paper we will consider such games on a certain class of transition systems, which we will call game structures.

Definition 1. A game structure is a tuple $(Q, Q_0, A, \longrightarrow)$ where:

- 1. Q is a finite set of states;
- 2. $\overline{Q}_0 \subseteq Q$ is a set of initial states;
- 3. A is a finite set of actions partitioned in two components A_c and A_e satisfying $A_c \cup A_e = A$ and $A_c \cap A_e = \emptyset$. Intuitively, the set A_c represents the set of controller actions while A_e represents the set of environment actions;
- 4. $\longrightarrow \subseteq Q \times A \times Q$ is a transition relation.

A game structure is said to be deterministic if $(q_1, a, q_2) \in \longrightarrow$ and $(q_1, a, q_3) \in \longrightarrow$ implies $q_2 = q_3$.

We will frequently resort to the more intuitive notation $q_1 \xrightarrow{a} q_2$ to represent $(q_1, a, q_2) \in \longrightarrow$. We will also restrict our attention to deterministic games where the actions of each player uniquely determine the next state. This is a natural assumption when the nondeterminism in the controller (environment) actions is due to environmental (controller) effects. However, the specification and the controller are allowed to be nondeterministic.

Note that the adopted game model does not require explicit alternation between controller and environment moves, neither does preclude it. However, controller and environment do not play simultaneously. This is simply a technical artifact, since we can consider their actions simultaneous if no information about the opponent move can be used at the time of play. Other game formulations consider game structures where simultaneous play is built in the transition relation as is the case in [13]. These game models, from now on called simultaneous, have a similar structure to the introduced game structures, except that $A = A_c + A_e$ is replaced by $A = A_c \times A_e$. Simultaneous game models $X' = (Q', Q'_0, A'_c \times A'_e, \longrightarrow)$ can be embedded in our framework resulting in games $NS(X') = X = (Q, Q_0, A, \longrightarrow)$ defined by:

- 1. $Q = Q' \cup Q' \times A'_c$;
- 2. $Q_0 = Q'_0$;

3.
$$A = A_c + A_e$$
, $A_c = A'_c$ and $A_e = A'_e$;

- 4. $q_1 \xrightarrow{a_c} q_2$ in X with $a_c \in A_c$ iff $q_2 = (q_1, a_c)$ and there is a state $q_3 \in Q$ and an action $a_e \in A_e$ such that $q_1 \xrightarrow{a_c, a_e} q_3$ in X';
- 5. $q_2 \xrightarrow{a_e} q_3$ in X with $a_e \in A_e$ iff $q_2 = (q_1, a_c), a_c \in A_c$ and $q_1 \xrightarrow{a_c, a_e} q_3$ in X':

We shall not elaborate on the properties of such embedding as it will only be used to relate the notions of alternating simulation and bisimulation introduced in [13] with the ones proposed in this paper. Before introducing such notions, we introduce morphisms between games so as to define the category where we shall develop our study of the control synthesis problem.

Definition 2. A morphism $f : X \to Y$ between two game structures X = $(Q_X, Q_{0X}, A_X, \longrightarrow)$ and $Y = (Q_Y, Q_{0Y}, A_Y, \longrightarrow)$ is given by a pair of maps $f = (f_Q, f_A)$ with $f_Q : Q_X \rightarrow Q_Y$ a totally defined map and $f_A : A_X \rightarrow A_Y$ a partially defined map satisfying:

- 1. $f_Q(Q_{0X}) \subseteq Q_{0Y};$
- 2. $f_A(A_{cX}) \subseteq A_{cY}$ and $f_A(A_{eX}) \subseteq A_{eY}$; 3. $q_1 \xrightarrow{a} q_2$ in X implies $f_Q(q_1) \xrightarrow{f_A(a)} f_Q(q_2)$ in Y if $f_A(a)$ is defined and $f_Q(q_1) = f_Q(q_2)$ if $f_A(a)$ is not defined.

It is not difficult to see that game structures with the above defined morphisms constitute a category. We shall denote such category by G. Furthermore, since our game models are in particular transition systems, the category G is, in many respects, similar to the category of transition systems introduced in [10] thus sharing many of its properties.

3 Bisimulation and Open Maps

In this section we quickly review the open maps framework introduced by Joyal and co-workers [9]. We consider a category **M** of machines with morphisms $X \xrightarrow{f} Y$ describing how machine *Y* simulates machine *X*. In this framework, the notion of bisimulation is introduced by resorting to the notion of computation path. We thus consider a subcategory **P** of **M** of path objects whose morphisms describe how paths objects can be extended.

To illustrate this approach we take G as the category of machines and for P we consider the full subcategory of G consisting of objects of the form:

$$q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} q_3 \xrightarrow{a_3} \dots \xrightarrow{a_{n-1}} q_n$$
 (1)

with q_1 as initial state and $q_i \neq q_j$ for $i \neq j$. We also define the control length of an object M of \mathbf{P} , denoted by $l_c(M)$, as the number of (not necessarily distinct) controller actions in (1). Similarly, the environment length of M, denoted by $l_e(M)$, is given by the number of environment actions in (1). Given two path objects M and N, a morphism $M \xrightarrow{\circ} N$ sends the initial state q_1 of M into the initial state q'_1 of N, the immediate successor of q_1 into the immediate successor of q'_1 and so on. We thus see that o only exists when $l_c(N) + l_e(N) \ge l_c(M) + l_e(M)$ in which case N can be seen as an extension of M. A game path in a game Xis now defined as a morphism from a path object M into X, that is $M \xrightarrow{m} X$. Intuitively, morphism $M \xrightarrow{m} X$ describes a possible evolution of the game modeled by X. A morphism $X \xrightarrow{f} Y$ between games can now be seen as describing how Y simulates the game evolution or path $M \xrightarrow{m} X$ by the game evolution path $M \xrightarrow{f \circ m} Y$.

Bisimulation is described through a special path lifting property:

Definition 3. A morphism $X \xrightarrow{f} Y$ is said to be **P**-open if given the left commutative diagram in (2), where M and N are path objects, there exists a diagonal morphism $N \xrightarrow{r} X$ making the right diagram in (2) commutative, that is, $m = r \circ o$ and $n = f \circ r$.

$$M \xrightarrow{m} X \qquad M \xrightarrow{m} X$$

$$\downarrow f \qquad \downarrow f \qquad \downarrow f \qquad \downarrow f \qquad \downarrow f \qquad (2)$$

$$N \xrightarrow{n} Y \qquad N \xrightarrow{n} Y$$

TEAM LING

In the category G with the above defined path category, the notion of P-open morphism admits the following characterization:

Proposition 1 (Adapted from [9]). A morphism $X \xrightarrow{f} Y$ is **P**-open iff for all reachable states q_1 of X:

if $f_Q(q_1) \xrightarrow{a'} q'_2$ in Y, then $q_1 \xrightarrow{a} q_2$ in X, $f_A(a') = a$ and $f_Q(q'_2) = q_2$.

We now consider the fiber subcategories \mathbf{G}_A and \mathbf{P}_A defined by the objects of \mathbf{G} and \mathbf{P} having the same action set A and morphisms f satisfying $f_A = \mathbf{1}_A$. In these subcategories we recover Park [21] and Milner's [22] notion of strong bisimulation through a span of \mathbf{P}_A -open maps:

Theorem 1 ([9]). Let X and Y be objects in \mathbf{G}_A . X is bisimilar to Y iff there exists a span $X \xleftarrow{f} Z \xrightarrow{g} Y$ with f a **P**-open morphism and g a **P**-open morphism.

In this setting, a deterministic game model X in G_A can be characterized by the existence of at most one morphism from a path object in P_A to X.

4 Alternating Simulation and Open Maps

To introduce alternating simulations we follow a similar route as the one outlined in the previous section by considering two path categories, one for each player:

Definition 4. The controller (environment) path category $\mathbf{P_c}$ ($\mathbf{P_e}$) consists of the objects of \mathbf{P} and morphisms $M \xrightarrow{o} N$ satisfying $l_c(N) \ge l_c(M)$ and $l_e(N) = l_e(M)$ ($l_c(N) = l_c(M)$ and $l_e(N) \ge l_e(M)$).

Note that when $l_c(N) \ge l_c(M)$ and $l_e(N) = l_e(M)$, path N extends path M only by controller moves and when $l_e(N) \ge l_e(M)$ and $l_c(N) = l_c(M)$, path N extends path M only by environment moves. Similarly to our discussion in Section 3 we have the following characterization of $\mathbf{P_e}$ -open and $\mathbf{P_c}$ -open morphisms which is a straightforward generalization of Proposition 1:

Proposition 2. Let $X \xrightarrow{f} Y$ be a morphism in G. Then, f is $\mathbf{P_c}$ -open ($\mathbf{P_e}$ -open) iff for any reachable state q_1 in X, $f_Q(q_1) \xrightarrow{a'} q'_2$ in Y implies $q_1 \xrightarrow{a} q_2$ in X, $f_A(a) = a'$ and $f_Q(q_2) = q'_2$ with $a \in A_{cX}$ ($a \in A_{eX}$).

The above result immediately suggests the following definition of controller and environment simulations:

Definition 5. Let X and Y be objects in G. Game X c-simulates (e-simulates) game Y if there exists a span $X \xleftarrow{g} Z \xrightarrow{h} Y$ with g a $\mathbf{P_e}$ -open ($\mathbf{P_c}$ -open) morphism and h a $\mathbf{P_c}$ -open ($\mathbf{P_e}$ -open) morphism.

The previous definition captures Alur and co-workers notion of alternating simulation [13] when two player simultaneous games are considered. For later use we recall such notion in this context:

Definition 6 (Adapted from [13]). Let $X = (Q_X, Q_{0X}, A_X, \longrightarrow)$ and $Y = (Q_Y, Q_{0Y}, A_Y, \longrightarrow)$ be simultaneous games. A relation $H \subseteq Q_X \times Q_Y$ is a *c*-simulation from X to Y if for all states $(q_{1x}, q_{1y}) \in H$ we have:

for every controller action $a_{cX} \in A_{cX}$ available at q_{1x} there exists a controller action $a_{cY} \in A_{cY}$ available at q_{1y} such that for every environment action $a_{eY} \in A_{eY}$ available at q_{1y} there is an environment action $a_{eX} \in A_{eX}$ available at q_{1x} satisfying $q_{1x} \xrightarrow{a_{cX}, a_{eX}} q_{2x}$ in $X, q_{1y} \xrightarrow{a_{eY}, a_{eY}} q_{2y}$ in Y and $(q_{2x}, q_{2y}) \in H$.

Environment simulations or *e*-simulations are obtained from controller simulations or *c*-simulations by reversing the role of the controller and environment. The precise equivalence between Definition 5 and Definition 6 is characterized in the following result:

Theorem 2. Let X and Y be two simultaneous game models and NS(X) and NS(Y) the corresponding objects in **G**. Then, NS(X) c-simulates (e-simulates) NS(Y), in the sense of Definition 5, iff X c-simulates (e-simulates) Y in the sense of Definition 6.

It is now clear that the notion of alternating simulation can be naturally described within the open maps framework. An interesting question not addressed in this paper is the study of alternating simulation notions induced by Definition 5 in other classes of concurrency models as well as the corresponding logic characterizations. Alternating simulation will play a fundamental role in the control synthesis problem described in the next section.

5 Control Synthesis

Co-fibrations and Parallel Composition. The control synthesis problem requires, in addition to bisimulations and alternating simulations, a notion of parallel composition. As detailed in [10], the usual notions of parallel composition can not be described by a single categorical construct. Instead, they are obtained by a sequence of product, restriction and relabeling operations. In this paper, we consider only the usual composition by synchronization on common events, although through a simpler alternative description resorting to co-fibrations. To motivate the notion of co-fibration, we revisit our game category G.

Every game model X contains a set of actions and every morphism f contains a map f_A transforming actions into actions. This suggests a "projection" functor V from **G** to the category of sets and partial maps between sets. Such functor has the obvious definition $V(X) = V(Q, Q_0, A, \longrightarrow) = A$ and $V(f) = V(f_Q, f_A) =$ f_A . For a given set A, we denote by \mathbf{G}_A the fiber category consisting of the objects X of **G** satisfying V(X) = A and morphisms f satisfying $V(f) = 1_A$. Consider now a morphism $X \xrightarrow{f} Y$ in **G** and let $V(X) = A_X$ and $V(Y) = A_Y$. We can construct an object $X^{\#}, V(X^{\#}) = V(Y)$, from X and f_A by replacing every $q_1 \xrightarrow{a} q_2$ in X with $q_1 \xrightarrow{f_A(a)} q_2$. This new object allows to factor f as $X \xrightarrow{f^{\#}} X^{\#} \xrightarrow{f} Y$, where $f^{\#} = (1_Q, f_A)$ and $\overline{f} = (f_Q, 1_{A_Y})$. Furthermore, for any other morphism $X \xrightarrow{g} Z$ with V(g) = V(f) there exists a unique morphism $X^{\#} \xrightarrow{\overline{g}} Z$ such that $\overline{g} \circ f^{\#} = f'$ as is pictorially represented in (3).



Such unique factorization properties are abstracted into the notion of cofibration that we now introduce following [23].

Definition 7. Let $F : \mathbf{D} \to \mathbf{E}$ be a functor and $\alpha : J \to I$ a morphism of \mathbf{E} . A morphism $f^{\#}: X \to Y$ of **D** is pre-cocartesian over α if:

- 1. $F(f^{\#}) = \alpha$; 2. $ifg: X \to Z$ is a morphism of **E** such that $F(g) = \alpha$, there exists a unique morphism in the fiber $\mathbf{D}_I h: Y \to Z$ such that $g = h \circ f^{\#}$

Pre-cocartesian morphisms are used to define co-fibrations as follows:

Definition 8. A functor $F : \mathbf{D} \to \mathbf{E}$ is said to be a co-fibration if:

- 1. for every morphism $\alpha: J \to I$ of **E** and every object X in the fiber over J, there exists in **D** a pre-cocartesian morphism $f^{\#}: X \to Y$ over α ;
- 2. the composition of two pre-cocartesian morphisms is again pre-cocartesian.

At this point the reader may find useful to return to diagram (3) and the discussion preceding it. Once again looking at G, we see that every pre-cocartesian morphism $f^{\#}$ is P-open, since every $1_Q(q_1) \xrightarrow{a'} q_2$ in $X^{\#}$ was obtained from a transition $q_1 \xrightarrow{a} q_2$ in X with $a' = f_A(a)$ which implies P-openness of f by Proposition 2. Based on this observation, we will make the following assumption which will hold throughout the paper:

A.I The game category **G** is equipped with a functor $V : \mathbf{G} \rightarrow \mathbf{L}$ which is a co-fibration. Furthermore, the co-fibration respects open maps in the sense that every pre-cocartesian morphism in G is P-open.

We now turn to another important ingredient, parallel composition. We shall abstract the usual notion of parallel composition by synchronization on common events to our framework through the following assumption:

A.II The parallel composition operator restricts to a fiber product, that is, for objects X and Y in the fiber \mathbf{G}_A , $X \parallel Y = X \times_A Y$. Furthermore, $X \parallel Y$ comes equipped with morphisms $X \xleftarrow{x} X \parallel Y \xrightarrow{y} Y$.

We now recall the definition of composition by synchronization on common events with the purpose of illustrating the above assumption.

Definition 9. Let X and Y be objects in G. The parallel composition of X and Y by synchronization on common events is the object $X \parallel Y = (Q_X \times Q_Y, Q_{0X} \times Q_{0Y}, (A_{cX} \cup A_{cY}) + (A_{eX} \cup A_{eY}), \longrightarrow)$ defined by $(q_{1x}, q_{1y}) \xrightarrow{a} (q_{2x}, q_{2y})$ in $X \parallel Y$ if:

1. $q_{1x} \xrightarrow{a} q_{2x}$ in X, $q_{1y} \xrightarrow{a} q_{2y}$ in Y and $a \in A_{cX} \cap A_{cY}$ or $a \in A_{eX} \cap A_{eY}$. 2. $q_{1x} \xrightarrow{a} q_{2x}$ in X, $q_{1y} = q_{2y}$ and $a \in A_{cX}$, $a \notin A_{cY}$ or $a \in A_{eX}$, $a \notin A_{eY}$. 3. $q_{1y} \xrightarrow{a} q_{2y}$ in Y, $q_{1x} = q_{2x}$ and $a \in A_{cY}$, $a \notin A_{cX}$ or $a \in A_{eY}$, $a \notin A_{eX}$.

This notion of parallel composition comes equipped with projection morphisms $X \stackrel{x}{\longleftarrow} X \parallel Y \stackrel{y}{\longrightarrow} Y$ defined by $x_Q(q_x, q_y) = q_x$, $x_A(a) = a$ if $a \in A_X$ and $x_A(a)$ undefined in $a \notin A_X$. Morphism y is similarly defined. Furthermore, when $A_X = A = A_Y$, $X \parallel Y$ coincides with the categorical product $X \times_A Y$ on the fiber category \mathbf{G}_A . Recall that the categorical product $X \times_A Y$ is the object of \mathbf{G}_A equipped with morphisms $X \stackrel{\pi_X}{\longleftarrow} X \times_A Y \stackrel{\pi_Y}{\longrightarrow} Y$ and satisfying the following property: for every $X \stackrel{f}{\longleftarrow} Z \stackrel{g}{\longrightarrow} Y$ in \mathbf{G}_A , there is one and only one morphism $h: Z \to X \times_A Y$ such that $\pi_X \circ h = f$ and $\pi_Y \circ h = g$.

Assumptions **A.I** and **A.II** provide a general setup allowing to study the control synthesis problem across several different categories of game or computation models. In addition to the working example of transition systems, in Section 6 we will apply the developed results to timed transition systems.

Existence and Synthesis of Controllers (Bisimulation). We now consider the control synthesis problem for bisimulation equivalence, that is, given a plant *P* and a specification *S* we seek to determine if a controller *C* rendering $C \parallel P$ bisimilar to *S* exists. More specifically we have:

Definition 10. Let P, S and C be objects in G. Object C is a bisimulation controller for plant P, enforcing specification S, if the following holds:

- 1. Morphism $p: C \parallel P \rightarrow P$ is $\mathbf{P_e}$ -open;
- 2. There exists a span $S \xleftarrow{s} Z \xrightarrow{cp} C \parallel P$ with s a **P**-open morphism and cp a **P**-open morphism, that is, S bisimulates $C \parallel P$.

The first condition requires controller C not to restrict environment moves as these cannot be influenced by the controller. The second condition asks for bisimulation equivalence between the controlled game $C \parallel P$ and the specification, a natural requirement in a branching time framework. Necessary and sufficient conditions for the existence of such controller can be formulated in terms of certain **P**-open and **P**_e-open morphisms:

Theorem 3. Let P be a deterministic object in **G** and S an arbitrary object in **G**. There exists a bisimulation controller C for plant P enforcing specification S iff there is a span $S \xleftarrow{\gamma} Z \xrightarrow{\delta} P$ with γ a **P**-open morphism and δ a **P**_e-open morphism. Furthermore, when a bisimulation controller C exists, we can take $C = Z^{\#}$ which has the same set of actions as P.

The previous result shows that existence of a bisimulation controller is equivalent to the requirement that P must simulate a bisimilar version Z of S while ensuring that every environment move in P is also possible in Z. This is a natural requirement as the controller C will restrict P to the image under δ of Z.

Existence and Synthesis of Controllers (*e***-Simulation)**.We now restrict attention to safety environment properties and liveness control specifications. These requirements are modeled by requiring the specification to *e*-simulate the controlled game. A controller enforcing the specification through an *e*-simulation restricts the effect of disturbances to accommodate safety properties while being as live as required by the specification. Formally, we define *e*-simulation controllers as follows:

Definition 11. Let P, S and C be objects in **G**. Object C is a *e-simulation* controller for plant P, enforcing specification S, if the following holds:

- 1. Morphism $p: C \parallel P \rightarrow P$ is $\mathbf{P_e}$ -open.
- 2. There exists a span $S \stackrel{s}{\longleftarrow} Z \stackrel{cp}{\longrightarrow} C \parallel P$ with $s \mid \mathbf{P_c}$ -open morphism and $cp \mid \mathbf{P_e}$ -open morphism, that is, $S \mid \mathbf{e}$ -simulates $C \parallel P$.

This kind of specification appears to be new since the Ramadge-Wonham framework only considers language equality, which corresponds to bisimulation in the branching time setting, or language inclusion which corresponds to simulation in the branching time setting. Simulation requirements are in fact weaker than *e*-simulation requirements and are discussed below.

Theorem 4. Let P be a deterministic object in **G** and S an arbitrary object in **G**. There exists an *e*-simulation controller C for plant P enforcing specification S iff there is a span: $S \xleftarrow{\gamma} Z \xrightarrow{\delta} P$ with γ a $\mathbf{P_c}$ -open morphism and δ a $\mathbf{P_e}$ -open morphism. Furthermore, when an *e*-simulation controller C exists, we can take $C = Z^{\#}$ which has the same set of actions as P.

It is interesting to note that, with respect to Theorem 3, only the assumptions of the left leg of span $S \xleftarrow{\gamma} Z \xrightarrow{\delta} P$ have been weakened. The same observation holds with respect to the results of the next section where a weaker version of the control synthesis problem is considered.

Existence and Synthesis of Controllers (Simulation). We now further weaken the control synthesis problem by only requiring the specification to simulate the controlled game. To illustrate the difference with respect an *e*-simulation requirement, we consider the specification, plant, controller and controlled system displayed in Figure 1. Controller *C* enforces the specification *S* by preventing the occurrence of action c_2 at the initial state. By looking at the controlled game $C \parallel P$ we see that there is an obvious inclusion morphism from $C \parallel P$ to *S* showing that *S* simulates the controlled game. However, *C* fails to be an *e*-simulation controller since it violates the liveness requirement to perform action c_2 at the initial state. Simulation requirements are therefore weaker than *e*-simulation requirements and constitute a natural specification when *e*-simulation controllers



Fig. 1. Pictorial representation of the plant *P*, specification *S*, controller *C* and corresponding controlled system $C \parallel P$

cannot be obtained. Nevertheless, requiring the specification only to simulate the controlled game may result in a trivial control synthesis problem since a controller preventing the occurrence of any controller action may constitute a solution. To rule out such trivial controllers we follow the Ramadge-Wonham approach by imposing a mild liveness restriction on the controller. We will require the possible controller to enforce the specification without creating blocking states on the controlled game. Such nonblocking assumption is formalized in our context through the notion of maximal paths.

Definition 12. Let X be an object in **G** and $o: O \to X$ a path in X. Path o is said to be maximal for X if given any other path $o': O' \to X$ such that $o' \circ m = o$, there is one and only one morphism $m': O' \to O$ satisfying $o \circ m' = o'$. A morphism $X \xrightarrow{f} Y$ is said to preserve maximal paths if for every maximal path $O \xrightarrow{o} X$, $O \xrightarrow{f \circ o} Y$ is also a maximal path.

Given the above definitions we consider a controller *C* nonblocking, when the morphism $p: C \parallel P \rightarrow P$ preserves maximal paths. This definition captures the supervisory control notion of nonblocking controller as shown in the next result.

Proposition 3. Let C and P be objects in G. Morphism $p : C \parallel P \rightarrow P$ preserves maximal paths iff for any reachable state q_1 in $C \parallel P$, $p_Q(q_1) \xrightarrow{a'} q'_2$ in P implies $q_1 \xrightarrow{a} q_2$ in $C \parallel P$.

We are now ready to formulate the simulation version of the control synthesis problem:

Definition 13. Let P, S and C be objects in G. Object C is a simulation controller for plant P, enforcing specification S, if the following holds:

- 1. Morphism $p: C \parallel P \rightarrow P$ is $\mathbf{P_e}$ -open and preserves maximal paths.
- 2. There exists a span $S \stackrel{s}{\longleftarrow} Z \stackrel{cp}{\longrightarrow} C \parallel P$ with cp a **P**-open morphism, that is, S simulates $C \parallel P$.

Theorem 5. Let P be a deterministic object in **G** and S an arbitrary object in **G**. There exists a simulation controller C for plant P enforcing specification S iff there is a span $S \xleftarrow{\gamma} Z \xrightarrow{\delta} P$ with δ a **P**_e-open morphism preserving maximal paths. Furthermore, when a simulation controller C exists, we can take $C = Z^{\#}$ which has the same set of actions as P.

Once again, only the assumptions on the left leg of span $S \xleftarrow{\gamma} Z \xrightarrow{\delta} P$ have been reduced to the requirement that γ is simply a morphism. On the other hand the new nonblocking requirement is now reflected on the maximal path preservation assumption. The simplicity of Theorems 3, 4 and 5 and their applicability to a large class of concurrency models illustrates the merit of the open maps approach. To further emphasize applicability, we describe in the next section how the developed results can be used with timed transition systems.

6 Timed Transition Systems

In this section we briefly outline how the presented results can also be used for timed transition systems control synthesis problems. Timed transition systems are transition systems enriched with timing information. They correspond to timed automata [24] without acceptance conditions or accepting states. By partitioning the action set into controller and environment actions we can also talk about timed games on timed game structures:

Definition 14. A timed game structure is a tuple $(Q, Q_0, A, \mathbb{T}, \longrightarrow)$ where:

- 1. Q is a finite set of states;
- 2. $Q_0 \subseteq Q$ is a finite set of initial states;
- 3. A is a finite set of actions partitioned in two components A_c and A_e satisfying $A_c \cup A_e = A$ and $A_c \cap A_e = \emptyset$. Intuitively, the set A_c represents the set of controller actions while A_e represents the set of environment actions;
- 4. \mathbb{T} is a finite set of clocks;
- 5. $\longrightarrow \subseteq Q \times A \times \Omega \times 2^{\mathbb{T}} \times Q$ is a transition relation where Ω is a clock constraint generated by the grammar $\Omega ::= c \sim t_1 | t_1 + c \sim t_2 | \Omega \wedge \Omega$ with $\sim \in \{\leq, <, \geq, >\}, c \in \mathbb{R}$ and t_1, t_2 clock variables.

We will resort to the more intuitive notation $q_1 \xrightarrow[\omega,\rho]{a} q_2$ to represent $(q_1, a, \omega, \rho, q_2) \in \longrightarrow$. Intuitively, the set of clocks records the passage of time which is then used to determine if and when a transition can be taken. Timing conditions on transitions are captured by clock constraints $\omega \in \Omega$. If we are using l clocks, then a clock constraint can be identified with a subset of $(\mathbb{R}_0^+)^l$, denoted by $[\omega]_{\mathbb{T}}$, representing the clock values satisfying the constraint. Given a function $g: \mathbb{T}_1 \to \mathbb{T}_2$ between two sets of clocks and a constraint ω on the clocks in \mathbb{T}_2 , we denote by $[\omega \circ g]_{\mathbb{T}_1}$ the constraint induced by ω on the clocks in \mathbb{T}_1 . By associating the discrete state $q_1 \in Q$ with the current value $t_1 \in \mathbb{R}^l$ of the clocks in \mathbb{T} , we obtain a configuration (q_1, t_1) . Sequences of configurations describe how the states of a given timed transition system evolve over time. Such sequences:

$$(q_0, t_0) \xrightarrow{a_1} (q_1, t_1) \xrightarrow{a_2} (q_2, t_2) \xrightarrow{a_3} \dots \xrightarrow{a_n} (q_n, t_n)$$

can take place when for each *i*, there exists a transition $q_{i-1} \xrightarrow[]{a_i} q_i$ in the timed game structure, the transition time satisfies the clock constraint $t_{i-1} + (\tau_i - \tau_i)$

¹ We denote by 1 the element of $(\mathbb{R}_0^+)^l$ in which every component is equal to 1.

 τ_{i-1}) $\mathbf{1} \in [\omega_i]_{\mathbb{T}}$ and the *j*th clock time $(t_i)_j$ is updated by $(t_i)_j = (t_{i-1})_j + \tau_i - \tau_{i-1}$ if $j \notin \rho_i$ or $(t_i)_j = 0$ if $j \in \rho_i$. To completely describe our category of timed game structures, we define morphisms following [25].

Definition 15. A morphism $f : X \to Y$ between two timed game structures $X = (Q_X, Q_{0X}, A_X, \mathbb{T}_X, \longrightarrow)$ and $Y = (Q_Y, Q_{0Y}, A_Y, \mathbb{T}_Y, \longrightarrow)$ is given by a pair of maps $f = (f_Q, f_T)$ with $f_Q : Q_X \to Q_Y$ and $f_T : \mathbb{T}_Y \to \mathbb{T}_X$ satisfying:

- 1. $f_Q(Q_{0X}) \subseteq Q_{0Y};$
- 2. $q_{1x} \xrightarrow[\omega_x,\rho_x]{\alpha} q_{2x}$ in X implies $f_Q(q_{1x}) \xrightarrow[\omega_y,\rho_y]{\alpha} f_Q(q_{2x})$ in Y with $\rho_y = f_{\mathbf{T}}^{-1}(\rho_x) = \{c \in \mathbb{T}_Y \mid f_{\mathbf{T}}(c) \in \rho_x\}$ and $[\omega_x \circ f_{\mathbf{T}}]_{\mathbf{T}_Y} \subseteq [\omega_y]_{\mathbf{T}_Y}$.

Note that we are only considering timed game structures with the same labeling set A and morphisms relating actions through the identity map on actions. This means that we are in fact working on the fiber subcategory over A. This also means that assumption A.I is automatically satisfied since pre-cocartesian morphisms are simply identity morphisms given the fact that $V(f) = 1_A$ for every morphism f. Assumption A.II is also satisfied as we will consider the categorical product between timed game structures as our notion of parallel composition [25]. The path subcategory P required to define bisimulation is now introduced through the use of timed words.

Definition 16. A timed word α over an alphabet A is an element of $(A \times \mathbb{R}_{\geq 0})^*$, that is, a finite sequence:

$$\alpha = (a_1, \tau_1)(a_2, \tau_2)(a_3, \tau_3) \dots (a_n, \tau_n)$$

satisfying $a_i \in A$, $\tau_i \in \mathbb{R}_{\geq 0}$ and $\tau_{i+1} > \tau_i$ for $1 \leq i \leq n$.

As detailed in [25], timed words can be embedded into **TG** as the following objects:

$$0 \xrightarrow{a_1}_{\omega_1,\rho_1} 1 \xrightarrow{a_2}_{\omega_2,\rho_2} 2 \xrightarrow{a_3}_{\omega_3,\rho_3} \dots \xrightarrow{a_n}_{\omega_n,\rho_n} n$$
(4)

where ω_i and ρ_i are appropriately chosen to create a full and faithful functor from the category of timed words and morphisms describing timed word extensions into **TG**. We refer the interested readers to [25] for the details of such embedding and consider **P** as the category of objects of the form (4) with morphisms describing how such objects can be extended. We also define the controller (environment) length of an object M in **P**, denoted by $l_c(M)$ ($l_e(M)$) as the number of not necessarily distinct controller (environment) actions appearing in M. Controller and environment lengths allow to define **P**_c and **P**_e as the subcategories of **P**, where morphisms $o: M \to N$ satisfy $l_e(M) = l_e(N)$ and $l_c(N) \ge l_c(N)$ when M and N are objects of **P**_c. Similarly to the un-timed case we only consider deterministic timed game structures.

With respect to definitions 10, 11 and 13, we now have the following characterization for the different control synthesis problems on timed game structures.

Theorem 6. Let P and S be objects in **TG**.

- 1. There exists a bisimulation controller C for plant P enforcing specification S iff there is a span $S \xleftarrow{\gamma} Z \xrightarrow{\delta} P$ with γ a **P**-open morphism and δ a **P**_e-open morphism.
- 2. There exists an *e-simulation* controller C for plant P enforcing specification S iff there is a span $S \stackrel{\gamma}{\longleftarrow} Z \stackrel{\delta}{\longrightarrow} P$ with γ a $\mathbf{P_c}$ -open morphism and δ a $\mathbf{P_e}$ -open morphism.
- 3. There exists a simulation controller C for plant P enforcing specification S iff there is a span $S \xleftarrow{\gamma} Z \xrightarrow{\delta} P$ with δ a $\mathbf{P_e}$ -open morphism preserving maximal paths.

Furthermore, when a bisimulation (e-simulation or simulation) controller C exists, we can take C = Z.

7 Future and Ongoing Work

We have only considered the control synthesis problem for deterministic systems. Determinism is a natural assumption when nondeterminism in the effect of controller (environment) actions is captured by the environment (controller) actions. However, nondeterminism may also exist due to other causes such as abstraction. It is therefore natural to extend the presented results to the nondeterministic case, especially since some of the proofs use determinism in a essential way. Other unexplored avenues include the instantiation of the developed results for other classes of systems such as Petri nets for which purely linear algebraic techniques [26] exist for controller synthesis. A different direction being currently investigated is the extension of the presented work to accommodate different notions of parallel composition.

References

- Ramadge, P., Wonham, W.M.: The control of discrete event systems. Proceedings of IEEE 77 (1989) 81–98
- de Alfaro, L., Henzinger, T.A.: Interface theories for component-based design. In: Proceedings of the First International Workshop on Embedded Software. Volume 2211 of Lecture Notes in Computer Science. (2001) 148–165
- Benveniste, A., Haar, S., Fabre, E., Jard, C.: Distributed monitoring of concurrent and asynchronous systems. In Amadio, R.M., Lugiez, D., eds.: Proceedings of the 14th International Conference on Concurrency Theory (CONCUR). Volume 2761 of Lecture Notes in Computer Science., Springer-Verlag (2003) 1–26
- 4. Tabuada, P., Pappas, G.J.: Linear time logic control of linear systems. (2004) Submitted for publication, available at www.nd.edu/~ptabuada.
- Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM Symposium on Principles of Programming Languages. (1989) 170–190
- Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In Press, I., ed.: Proceedings of the 31st Annual Symposium on Foundations of Computer Sience, St. Louis, Missouri (1990) 746–757

- 7. Emerson, E.A., Clarke, E.M.: Using branching time temporal logic to synthesize synchronization skeletons. Science of Computer Programming 2 (1982) 241–266
- Manna, Z., Wolper, P.: Synthesis of communication processes from temporal logic specifications. ACM Transactions on Programming Languages and Systems 6 (1984) 68–93
- 9. Joyal, A., Nielsen, M., Winskel, G.: Bisimulation from open maps. Information and Computation 127 (1996) 164–185
- Winskel, G., Nielsen, M.: Models for concurrency. In Abramsky, Gabbay, Maibaum, eds.: Handbook of Logic and Foundations of Theoretical Computer Science. Volume 4. Oxford University Press, London (1994)
- Nielsen, M., Cheng, A.: Observing behaviour categorically. In Thiagarajan, P.S., ed.: Foundations of Software Technology and Theoretical Computer Science. Volume 1026 of Lecture Notes in Computer Science., Springer (1995) 263–278
- Haghverdi, E., Tabuada, P., Pappas, G.: Bisimulation relations for dynamical and control systems. In Blute, R., Selinger, P., eds.: Electronic Notes in Theoretical Computer Science. Volume 69., Elsevier (2003)
- Alur, R., Henzinger, T., Kupferman, O., Vardi, M.: Alternating refinement relations. In: CONCUR 97: Concurrency Theory, 8th International Conference. Number 1466 in Lecture Notes in Computer Science (1998) 163–178
- 14. Madhusudan, P., Thiagarajan, P.: Branching time controllers for discrete event systems. Theoretical Computer Science **274** (2002) 117–149
- Barret, G., Lafortune, S.: Bisimulation, the supervisor control problem and strong model matching for finite state machines. Journal of Discrete Event Systems 8 (1998) 337–429
- 16. Rutten, J.: Coalgebra, concurrency, and control. In Kluwer, ed.: Proceedings of the 5th Workshop on Discrete Event Systems (WODES 2000). (2000) 31–38
- Overkamp, A.: Supervisory control using failure semantics and partial specifications. IEEE Transactions on Automatic Control 42 (1997) 498–510
- Kupferman, O., Madhusudan, P., Thiagarajan, P.S., Vardi, M.Y.: Open systems in reactive environments: Control and synthesis. In: Proceedings of the 11th International Conference on Concurrency Theory. Volume 1877 of Lecture Notes in Computer Science., Springer-Verlag (2000) 92–107
- Antoniotti, M., Mishra, B.: NP-completeness of the supervisor synthesis problem for unrestricted CTL specifications. In Smedinga, R., Spathopoulos, M., Kozk, P., eds.: Proceedings on the International Workshop on Discrete Event Systems, WODES96, Edinburgh, Scotland, UK (1996)
- Shengbing, J., Kumar, R.: Supervisory control of discrete event systems with CTL* temporal logic specifications. In: Proceedings of the 40th IEEE Conference on Decision and Control. Volume 5. (2001) 4122–4127
- Park, D.: Concurrency and automata on infinite sequences. Volume 104 of Lecture Notes in Computer Science. (1981) 167–183
- 22. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
- 23. Borceux, F.: Handbook of Categorical Algebra. Cambridge University Press (1994)
- Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126 (1994) 183–235
- Nielsen, M., Hune, T.S.: Bisimulation and open maps for timed transition systems. Fundamenta Informaticae 38 (1999) 61–77
- 26. Moody, J.O., Antsaklis, P.J.: Supervisory Control of Discrete Event Systems Using Petri Nets. Kluwer Academic Publishers (1998)

Probabilistic Event Structures and Domains

Daniele Varacca¹*, Hagen Völzer², and Glynn Winskel³

¹ LIENS - École Normale Supérieure, France

² Institut für Theoretische Informatik - Universität zu Lübeck, Germany
 ³ Computer Laboratory - University of Cambridge, UK

Abstract. This paper studies how to adjoin probability to event structures, leading to the model of probabilistic event structures. In their simplest form probabilistic choice is localised to cells, where conflict arises; in which case probabilistic independence coincides with causal independence. An application to the semantics of a probabilistic CCS is sketched. An event structure is associated with a domain-that of its configurations ordered by inclusion. In domain theory probabilistic processes are denoted by continuous valuations on a domain. A key result of this paper is a representation theorem showing how continuous valuations on the domain of a confusion-free event structure correspond to the probabilistic event structures it supports. We explore how to extend probability to event structures which are not confusion-free via two notions of probabilistic runs of a general event structure. Finally, we show how probabilistic correlation and probabilistic event structures with confusion can arise from event structures which are originally confusion-free by using morphisms to rename and hide events.

1 Introduction

There is a central divide in models for concurrent processes according to whether they represent parallelism by nondeterministic interleaving of actions or directly as causal independence. Where a model stands with respect to this divide affects how probability is adjoined. Most work has been concerned with probabilistic interleaving models [LS91,Seg95,DEP02]. In contrast, we propose a probabilistic causal model, a form of probabilistic event structure.

An event structure consists of a set of events with relations of causal dependency and conflict. A configuration (a state, or partial run of the event structure) consists of a subset of events which respects causal dependency and is conflict free. Ordered by inclusion, configurations form a special kind of Scott domain [NPW81].

The first model we investigate is based on the idea that all conflict is resolved probabilistically and locally. This intuition leads us to a simple model based on

© Springer-Verlag Berlin Heidelberg 2004

^{*} Work partially done as PhD student at BRICS - Aarhus, Denmark Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 481-496, 2004.

confusion-free event structures, a form of concrete data structures [KP93], but where computation proceeds by making a probabilistic choice as to which event occurs at each currently accessible cell. (The probabilistic event structures which arise are a special case of those studied by Katoen [Kat96]—though our concentration on the purely probabilistic case and the use of cells makes the definition simpler.) Such a probabilistic event structure immediately gives a "probability" weighting to each configuration got as the product of the probabilities of its constituent events. We characterise those weightings (called *configuration valuations*) which result in this way. Understanding the weighting as a true probability will lead us later to the important notion of probabilistic test.

Traditionally, in domain theory a probabilistic process is represented as a continuous valuation on the open sets of a domain, i.e., as an element of the probabilistic powerdomain of Jones and Plotkin [JP89]. We reconcile probabilistic event structures with domain theory, lifting the work of [NPW81] to the probabilistic case, by showing how they determine continuous valuations on the domain of configurations. In doing so however we do not obtain all continuous valuations. We show that this is essentially for two reasons: in valuations probability can "leak" in the sense that the total probabilistic choices at different cells need not be probabilistically independent. In the process we are led to a more general definition of probabilistic event structure from which we obtain a key representation theorem: continuous valuations on the domain of configurations correspond to the more general probabilistic event structures.

How do we adjoin probabilities to event structures which are not necessarily confusion-free? We argue that in general a probabilistic event structure can be identified with a probabilistic run of the underlying event structure and that this corresponds to a probability measure over the maximal configurations. This sweeping definition is backed up by a precise correspondence in the case of confusion-free event structures. Exploring the operational content of this general definition leads us to consider probabilistic tests comprising a set of finite configurations which are both mutually exclusive and exhaustive. Tests do indeed carry a probability distribution, and as such can be regarded as finite probabilistic partial runs of the event structure.

Finally we explore how phenomena such as probabilistic correlation between choices and confusion can arise through the hiding and relabeling of events. To this end we present some preliminary results on "tight" morphisms of event structures, showing how, while preserving continuous valuations, they can produce such phenomena.

2 Probabilistic Event Structures

2.1 Event Structures

An event structure is a triple $\mathcal{E} = \langle E, \leq, \# \rangle$ such that

- *E* is a countable set of *events*;
- ⟨E, ≤⟩ is a partial order, called the *causal order*, such that for every e ∈ E, the set of events ↓ e is finite;
- # is an irreflexive and symmetric relation, called the *conflict relation*, satisfying the following: for every $e_1, e_2, e_3 \in E$ if $e_1 \leq e_2$ and $e_1 \# e_3$ then $e_2 \# e_3$.

Causal dependence and conflict are mutually exclusive. If two events are not causally dependent nor in conflict they are said to be *concurrent*.

A configuration x of an event structure \mathcal{E} is a conflict-free downward closed subset of E, i.e. a subset x of E satisfying: (1) whenever $e \in x$ and $e' \leq e$ then $e' \in x$ and (2) for every $e, e' \in x$, it is not the case that e # e'. Therefore, two events of a configuration are either causally dependent or concurrent, i.e., a configuration represents a run of an event structure where events are partially ordered. The set of configurations of \mathcal{E} , partially ordered by inclusion, is denoted as $\mathcal{L}(\mathcal{E})$. The set of finite configurations is written by $\mathcal{L}_{fin}(\mathcal{E})$. We denote the empty configuration by \perp .

If x is a configuration and e is an event such that $e \notin x$ and $x \cup \{e\}$ is a configuration, then we say that e is *enabled* at x. Two configurations x, x' are said to be *compatible* if $x \cup x'$ is a configuration. For every event e of an event structure \mathcal{E} , we define $[e] := \downarrow e$, and $[e] := [e] \setminus \{e\}$. It is easy to see that any event e is enabled at [e].

We say that events e_1 and e_2 are in *immediate* conflict, and write $e_1 \#_{\mu} e_2$ when $e_1 \# e_2$ and both $[e_1] \cup [e_2]$ and $[e_1] \cup [e_2)$ are configurations. Note that the immediate conflict relation is symmetric. It is also easy to see that a conflict $e_1 \# e_2$ is immediate if and only if there is a configuration where both e_1 and e_2 are enabled.

2.2 Confusion-Free Event Structures

The most intuitive way to add probability to an event structure is to identify "probabilistic events", such as coin flips, where probability is associated locally. A probabilistic event can be thought of as probability distribution over a *cell*, that is, a set of events (the outcomes) that are pairwise in immediate conflict and that have the same set of causal predecessors. The latter implies that all outcomes are enabled at the same configurations, which allows us to say that the probabilistic event is either enabled or not enabled at a configuration.

Definition 2.1. A partial cell is a set c of events such that $e, e' \in c$ implies $e \#_{\mu} e'$ and [e] = [e']. A maximal partial cell is called a cell.

We will now restrict our attention to event structures where each immediate conflict is resolved through some probabilistic event. That is, we assume that cells are closed under immediate conflict. This implies that cells are pairwise disjoint.

Definition 2.2. An event structure is confusion-free if its cells are closed under *immediate conflict*.

Proposition 2.3. An event structure is confusion-free if and only if the reflexive closure of immediate conflict is transitive and inside cells, the latter meaning that $e \#_{\mu} e' \Longrightarrow [e] = [e')$.

It follows that, in a confusion-free event structure, the reflexive closure of immediate conflict is an equivalence with cells being its equivalence classes. If an event $e \in c$ is enabled at a configuration x, all the events of c are enabled as well. In which case we say that the cell c is *accessible* at x. Confusion-free event structures correspond to deterministic concrete data structures [NPW81,KP93] and to confusion-free occurrence nets [NPW81].

We find it useful to define cells without directly referring to events. To this end we introduce the notion of *covering*.

Definition 2.4. Given two configurations $x, x' \in \mathcal{L}(\mathcal{E})$ we say that x' covers x if there exists $e \notin x$ such that $x' = x \cup \{e\}$. For every finite configuration x of a confusion-free event structure, a partial covering at x is a set of pairwise incompatible configurations that cover x. A covering at x is a maximal partial covering at x.

Proposition 2.5. In a confusion-free event structure if C is a covering at x, then $c = \{e \mid x \cup \{e\} \in C\}$ is a cell accessible at x. Conversely, if c is accessible at x, then $C := \{x \cup \{e\} \mid e \in c\}$ is a covering at x.

2.3 Probabilistic Event Structures with Independence

Once an event structure is confusion-free, we can associate a probability distribution with each cell. Intuitively it is as if we have a die local to each cell, determining the probability with which the events at that cell occur. In this way we obtain our first definition of a probabilistic event structure, a definition in which dice at different cells are assumed probabilistically independent.

Definition 2.6. When $f: X \to [0, +\infty]$ is a function, for every $Y \subseteq X$, we define $f[Y] := \sum_{x \in Y} f(x)$. A cell valuation on a confusion-free event structure $\langle E, \leq, \# \rangle$ is a function $p: E \to [0, 1]$ such that for every cell c, we have p[c] = 1.

Assuming probabilistic independence of all probabilistic events, every finite configuration can be given a "probability" which is obtained as the product of probabilities of its constituent events. This gives us a function $\mathcal{L}_{fin}(\mathcal{E}) \rightarrow [0, 1]$ which we can characterise in terms of the order-theoretic structure of $\mathcal{L}_{fin}(\mathcal{E})$ by using coverings.

Proposition 2.7. Let p be a cell valuation and let $v : \mathcal{L}_{fin}(\mathcal{E}) \to [0,1]$ be defined by $v(x) = \prod_{e \in x} p(e)$. Then we have

- (a) (Normality) $v(\perp) = 1;$
- (b) (Conservation) if C is a covering at x, then v[C] = v(x);
- (c) (Independence) if x, y are compatible, then $v(x) \cdot v(y) = v(x \cup y) \cdot v(x \cap y)$.

Definition 2.8. A configuration valuation with independence on a confusionfree event structure \mathcal{E} is a function $v : \mathcal{L}_{fin}(\mathcal{E}) \to [0,1]$ that satisfies normality, conservation and independence. The configuration valuation associated with a cell valuation p as in Prop. 2.7 is denoted by v_p .

Proposition 2.9. If v is a configuration valuation with independence and p: $E \to [0,1]$ is a mapping such that $v([e]) = p(e) \cdot v([e))$ for all $e \in E$, then p is a cell valuation such that $v_p = v$.

Condition (c) from Proposition 2.7 is essential to prove Proposition 2.9. We will show later (Theorem 5.3) the sense in which this condition amounts to probabilistic independence.

We give an example. Take the following confusion-free event structure \mathcal{E}_1 : $E_1 = \{a, b, c, d\}$ with the discrete causal ordering and with $a \#_{\mu} b$ and $c \#_{\mu} d$.

We define a cell valuation on \mathcal{E}_1 by p(a) = 1/3, p(b) = 2/3, p(c) = 1/4, p(d) = 3/4. The corresponding configuration valuation is defined as

- $v_p(\perp) = 1;$
- $v_p(\{a\}) = 1/3, v_p(\{b\}) = 2/3, v_p(\{c\}) = 1/4, v_p(\{d\}) = 3/4;$
- $v_p(\{a,c\}) = 1/12, v_p(\{b,c\}) = 1/6, v_p(\{a,d\}) = 1/4, v_p(\{b,d\}) = 1/2.$

In the event structure above, a covering at \bot consists of $\{a\}, \{b\}$, while a covering at $\{a\}$ consists of $\{a, c\}, \{a, d\}$.

We conclude this section with a definition of a probabilistic event structure. Though, as the definition indicates, we will consider a more general definition later, one in which there can be probabilistic correlations between the choices at different cells.

Definition 2.10. A probabilistic event structure with independence *consists* of a confusion-free event structure together with a configuration valuation with independence.

3 A Process Language

Confusion-freeness is a strong requirement. But it is still possible to give a semantics to a fairly rich language for probabilistic processes in terms of probabilistic event structures with independence. The language we sketch is a probabilistic version of value passing CCS. Following an idea of Milner, used in the context of confluent processes [Mil89], we restrict parallel composition so that there is no ambiguity as to which two processes can communicate at a channel; parallel composition will then preserve confusion-freeness.

Assume a set of channels *L*. For simplicity we assume that a common set of values *V* may be communicated over any channel $a \in L$. The syntax of processes is given by:

$$P ::= 0 \mid \sum_{v \in V} a!(p_v, v).P_v \mid a?(x).P \mid P_1 ||P_2 \mid P \setminus A \mid$$
$$P[f] \mid \text{if } b \text{ then } P_1 \text{ else } P_2 \mid X \mid \text{rec } X.P$$

TEAM LING

Here *x* ranges over value variables, *X* over process variables, *A* over subsets of channels and *f* over injective renaming functions on channels, *b* over boolean expressions (which make use of values and value variables). The coefficients p_v are real numbers such that $\sum_{v \in V} p_v = 1$.

A closed process will denote a probabilistic event structure with independence, but with an additional labelling function from events to output labels a!v, input labels a?v where a is a channel and v a value, or τ . At the cost of some informality we explain the probabilistic semantics in terms of CCS constructions on the underlying labelled event structures, in which we treat pairs of labels consisting of an output label a!v and input label a?v as complementary. (See e.g. the handbook chapter [WN95] or [Win82,Win87] for an explanation of the event structure semantics of CCS.) For simplicity we restrict attention to the semantics of closed process terms.

The nil process 0 denotes the empty probabilistic event structure. A closed output process $\sum_{v \in V} a!(p_v, v).P_v$ can perform a synchronisation at channel a, outputting a value v with probability p_v , whereupon it resumes as the process P_v . Each P_v , for $v \in V$, will denote a labelled probabilistic event structure with underlying labelled event structure $\mathcal{E}[P_v]$. The underlying event structure of such a closed output process is got by the juxtaposition of the family of prefixed event structures

$a!v.\mathcal{E}\llbracket P_v rbracket$,

with $v \in V$, in which the additional prefixing events labelled a!v are put in (immediate) conflict; the new prefixing events labelled a!v are then assigned probabilities p_v to obtain the labelled probabilistic event structure.

A closed input process a?(x).P synchronises at channel a, inputting a value v and resuming as the closed process P[v/x]. Such a process P[v/x] denotes a labelled probabilistic event structure with underlying labelled event structure $\mathcal{E}[P[v/x]]$. The underlying labelled event structure of the input process is got as the parallel juxtaposition of the family of prefixed event structures

$$a?v.\mathcal{E}\llbracket P[v/x]
rbracket$$
,

with $v \in V$; the new prefixing events labelled a?v are then assigned probabilities 1.

The probabilistic parallel composition corresponds to the usual CCS parallel composition followed by restricting away on all channels used for communication. In order for the parallel composition $P_1 || P_2$ to be well formed the set of input channels of P_1 and P_2 must be disjoint, as must be their output channels. So, for instance, it is not possible to form the parallel composition

$$\sum_{v \in V} a!(p_v,v).0\|a?(x).P_1\|a?(x).P_2 \ .$$

In this way we ensure that no confusion is introduced through synchronisation.

We first describe the effect of the parallel composition on the underlying event structures of the two components, assumed to be E_1 and E_2 . This is got by CCS parallel composition followed by restricting away events in a set S:

 $(E_1 \,|\, E_2) \setminus S$

where S consists of all labels a!v, a?v for which a!v appears in E_1 and a?v in E_2 , or vice versa. In this way any communication between E_1 and E_2 is forced when possible. The newly introduced τ -events, corresponding to a synchronisation between an a!v-event with probability p_v and an a?v-event with probability 1, are assigned probability p_v .

A restriction $P \setminus A$ has the effect of the CCS restriction

 $\mathcal{E}[\![P]\!] \setminus \{a!v,a?v \mid v \in V \ \& \ a \in A\}$

on the underlying event structure; the probabilities of the events which remain stay the same. A renaming P[f] has the usual effect on the underlying event structure, probabilities of events being maintained. A closed conditional (if b then P_1 else P_2) has the denotation of P_1 when b is true and of P_2 when b is false.

The recursive definition of probabilistic event structures follows that of event structures [Win87] carrying the extra probabilities along. Though care must be taken to ensure that a confusion-free event structure results: one way to ensure this is to insist that for rec X.P to be well-formed the process variable X may not occur under a parallel composition.

4 Probabilistic Event Structures and Domains

The configurations $\langle \mathcal{L}(\mathcal{E}), \subseteq \rangle$ of a confusion-free event structure \mathcal{E} , ordered by inclusion, form a domain, specifically a *distributive concrete domain* (cf. [NPW81,KP93]). In traditional domain theory, a probabilistic process is denoted by a *continuous valuation*. Here we show that, as one would hope, every probabilistic event structure with independence corresponds to a unique continuous valuation. However not all continuous valuations arise in this way. Exploring why leads us to a more liberal notion of a configuration valuation, in which there may be probabilistic correlation between cells. This provides a representation of the normalised continuous valuations on distributive concrete domains in terms of probabilistic event structures. (The Appendix includes a brief survey of the domain theory we require. The rather involved proofs of this section can be found in [Var03].)

4.1 Domains

The probabilistic powerdomain of Jones and Plotkin [JP89] consists of continuous valuations, to be thought of as denotations of probabilistic processes. A *continuous valuation* on a DCPO D is a function ν defined on the Scott open subsets of D, taking values on $[0, +\infty]$, and satisfying:

- (Strictness) $\nu(\emptyset) = 0;$
- (Monotonicity) $U \subseteq V \Longrightarrow \nu(U) \le \nu(V);$
- (Modularity) $\nu(U) + \nu(V) = \nu(U \cup V) + \nu(U \cap V);$
- (Continuity) if \mathcal{J} is a directed family of open sets, $\nu(\bigcup \mathcal{J}) = \sup_{U \in \mathcal{J}} \nu(U)$.

A continuous valuation ν is *normalised* if $\nu(D) = 1$. Let $\mathcal{V}^1(D)$ denote the set of normalised continuous valuations on D equipped with the pointwise order: $\nu \leq \xi$ if for all open sets U, $\nu(U) \leq \xi(U)$. $\mathcal{V}^1(D)$ is a DCPO [JP89,Eda95].

The open sets in the Scott topology represent observations. If D is an algebraic domain and $x \in D$ is compact, the *principal* set $\uparrow x$ is open. Principal open sets can be thought of as basic observations. Indeed they form a basis of the Scott topology.

Intuitively a normalised continuous valuation ν assigns probabilities to observations. In particular we could think of the probability of a principal open set $\uparrow x$ as representing the probability of x.

4.2 Continuous and Configuration Valuations

As can be hoped, a configuration valuation with independence on a confusionfree event structure \mathcal{E} corresponds to a normalised continuous valuation on the domain $\langle \mathcal{L}(\mathcal{E}), \subseteq \rangle$, in the following sense.

Proposition 4.1. For every configuration valuation with independence v on \mathcal{E} there is a unique normalised continuous valuation v on $\mathcal{L}(\mathcal{E})$ such that for every finite configuration x, $v(\uparrow x) = v(x)$.

Proof. The claim is a special case of the subsequent Theorem 4.4. \Box

While a configuration valuation with independence gives rise to a continuous valuation, not every continuous valuation arises in this way. As an example, consider the event structure \mathcal{E}_1 as defined in Section 2.3. Define

- $\nu(\uparrow\{a\}) = \nu(\uparrow\{b\}) = \nu(\uparrow\{c\}) = \nu(\uparrow\{d\}) = 1/2;$
- $\nu(\uparrow\{a,d\}) = \nu(\uparrow\{b,c\}) = 1/2;$
- $\nu(\uparrow\{a,c\}) = \nu(\uparrow\{b,d\}) = 0;$

and extend it to all open sets by modularity. It is easy to verify that it is indeed a continuous valuation on $\mathcal{L}(\mathcal{E}_1)$. Define a function $v : \mathcal{L}_{\text{fin}}(\mathcal{E}_1) \to [0, 1]$ by $v(x) := v(\uparrow x)$. This is not a configuration valuation with independence; it does not satisfy condition (c) of Proposition 2.7. If we consider the compatible configurations $x := \{a\}, y := \{c\}$ then $v(x \cup y) \cdot v(x \cap y) = 0 < 1/4 = v(x) \cdot v(y)$.

Also continuous valuations "leaking" probability do not arise from probabilistic event structures with independence.

Definition 4.2. Denote the set of maximal elements of a DCPO D by $\Omega(D)$. A normalised continuous valuation ν on D is non-leaking if for every open set $O \supseteq \Omega(D)$, we have $\nu(O) = 1$. This definition is new, although inspired by a similar concept in [Eda95]. For the simplest example of a leaking continuous valuation, consider the event structure \mathcal{E}_2 consisting of one event *e* only, and the valuation defined as $\nu(\emptyset) = 0$, $\nu(\uparrow \bot) = 1$, $\nu(\uparrow \{e\}) = 1/2$. The corresponding function $v : \mathcal{L}_{\text{fin}}(\mathcal{E}_2) \to [0, 1]$ violates condition (*b*) of Proposition 2.7. The probabilities in the cell of *e* do not sum up to 1.

We analyse how valuations without independence and leaking valuations can arise in the next two sections.

4.3 Valuations Without Independence

Definition 2.10 of probabilistic event structures assumes the probabilistic independence of choice at different cells. This is reflected by condition (c) in Proposition 2.7 on which it depends. In the first example above, the probabilistic choices in the two cells are not independent: once we know the outcome of one of them, we also know the outcome of the other. This observation leads us to a more general definition of a configuration valuation and probabilistic event structure.

Definition 4.3. A configuration valuation on a confusion-free event structure \mathcal{E} is a function $v : \mathcal{L}_{fin}(\mathcal{E}) \to [0, 1]$ such that:

(a) $v(\perp) = 1$; (b) if C is a covering at x, then v[C] = v(x).

A probabilistic event structure consists of a confusion-free event structure together with a configuration valuation.

Now we can generalise Proposition 4.1, and provide a converse:

Theorem 4.4. For every configuration valuation v on \mathcal{E} there is a unique normalised continuous valuation v on $\mathcal{L}(\mathcal{E})$ such that for every finite configuration x, $\nu(\uparrow x) = v(x)$. Moreover v is non-leaking.

Theorem 4.5. Let ν be a non-leaking continuous valuation on $\mathcal{L}(\mathcal{E})$. The function $v : \mathcal{L}_{\text{fin}}(\mathcal{E}) \to [0,1]$ defined by $v(x) = \nu(\uparrow x)$ is a configuration valuation.

The two theorems above provide a representation of non-leaking continuous valuations on distributive concrete domains—see [Var03], Thm. 6.4.1 and Thm. 7.6.2 for their proof. Using this representation result, we are also able to characterise the maximal elements in $\mathcal{V}^1(\mathcal{L}(\mathcal{E}))$ as precisely the non-leaking valuations—a fact which is not known for general domains.

Theorem 4.6. Let \mathcal{E} be a confusion-free event structure and let $\nu \in \mathcal{V}^1(\mathcal{L}(\mathcal{E}))$. Then ν is non-leaking if and only if it is maximal.

Proof. See [Var03], Prop. 7.6.3 and Thm. 7.6.4.
4.4 Leaking Valuations

There remain leaking continuous valuations, as yet unrepresented by any probabilistic event structures. At first sight it might seem that to account for leaking valuations it would be enough to relax condition (b) of Definition 4.3 to the following

(b') if C is a covering at x, then $v[C] \leq v(x)$.

However, it turns out that this is not the right generalisation, as the following example shows. Consider the event structure \mathcal{E}_3 where $E_3 = \{a, b\}$ with the flat causal ordering and no conflict. Define a "leaking configuration valuation" on \mathcal{E}_3 by $v(\perp) = v(\{a\}) = v(\{b\}) = 1, v(\{a, b\}) = 0.$

The function v satisfies conditions (a) and (b'), but it cannot be extended to a continuous valuation on the domain of configurations. However, we can show that the leaking of probability is attributable to an "invisible" event.

Definition 4.7. Consider a confusion-free event structure $\mathcal{E} = \langle E, \leq, \# \rangle$. For every cell c we consider a new "invisible" event ∂_c such that $\partial_c \notin E$ and if $c \neq c'$ then $\partial_c \neq \partial_{c'}$. Let $\partial = \{\partial_c \mid c \text{ is a cell}\}$. We define \mathcal{E}_{∂} to be $\langle E_{\partial}, \leq_{\partial}, \#_{\partial} \rangle$, where

- $E_{\partial} = E \cup \partial$;
- ≤_∂ is ≤ extended by e ≤_∂ ∂_c if for all e' ∈ c, e ≤ e';
 #_∂ is # extended by e #_∂ ∂_c if there exists e' ∈ c, e' ≤ e.

So \mathcal{E}_{∂} is \mathcal{E} extended by an extra invisible event at every cell. Invisible events can absorb all leaking probability, as shown by Theorem 4.9 below.

Definition 4.8. Let \mathcal{E} be a confusion-free event structure. A generalised configuration valuation on \mathcal{E} is a function $v: \mathcal{L}_{fin}(\mathcal{E}) \to [0,1]$ that can be extended to a configuration valuation on \mathcal{E}_{∂} .

It is not difficult to prove that, when such an extension exists, it is unique.

Theorem 4.9. Let \mathcal{E} be a confusion-free event structure. Let $v : \mathcal{L}_{fin}(\mathcal{E}) \to [0,1]$. There exists a unique normalised continuous valuation ν on $\mathcal{L}(\mathcal{E})$ with v(x) = $\nu(\uparrow x)$, if and only if v is a generalised configuration valuation.

Proof. See [Var03], Thm. 6.5.3.

The above theorem completely characterises the normalised continuous valuations on distributive concrete domains in terms of probabilistic event structures.

Probabilistic Event Structures as Probabilistic Runs 5

In the rest of the paper we investigate how to adjoin probabilities to event structures which are not confusion-free. In order to do so, we find it useful to introduce two notions of probabilistic run.

Configurations represent runs (or computation paths) of an event structure. What is a probabilistic run (or probabilistic computation path) of an event

structure? One would expect a probabilistic run to be a form of probabilistic configuration, so a probability distribution over a suitably chosen subset of configurations. As a guideline we consider the traditional model of probabilistic automata [Seg95], where probabilistic runs are represented in essentially two ways: as a probability measure over the set of maximal runs [Seg95], and as a probability distribution over finite runs of the same length [dAHJ01].

The first approach is readily available to us, and where we begin. As we will see, according to this view probabilistic event structures over an underlying event structure \mathcal{E} correspond precisely to the probabilistic runs of \mathcal{E} .

The proofs of the results in this section are omitted, but they can be found in the technical report [VVW04].

5.1 **Probabilistic Runs of an Event Structure**

The first approach suggests that a probabilistic run of an event structure \mathcal{E} be taken to be a probability measure on the maximal configurations of $\mathcal{L}(\mathcal{E})$.

To do so requires some notions from measure theory. A measurable space is a pair $\langle \Omega, S \rangle$, where Ω is a set and S is a σ -algebra over Ω . A measure over a measurable space $\langle \Omega, S \rangle$ is a countably additive function $\mu : S \to [0, +\infty]$. If $\mu(\Omega) = 1$, we talk of a probability measure. Let D be an algebraic domain. Recall that $\Omega(D)$ denotes the set of maximal elements of D and that for every compact element $x \in D$ the *principal* set $\uparrow x$ is Scott open. The set K(x) := $\uparrow x \cap \Omega(D)$ is called the *shadow* of x. We shall consider the σ -algebra S on $\Omega(D)$ generated by the shadows of the compact elements. The configurations of an event structure form a coherent ω -algebraic domain, whose compact elements are the finite configurations [NPW81].

Definition 5.1. A probabilistic run of an event structure \mathcal{E} is a probability measure on $\langle \Omega(\mathcal{L}(\mathcal{E})), \mathcal{S} \rangle$, where \mathcal{S} is the σ -algebra generated by the shadows of the compact elements.

There is a tight correspondence between non-leaking valuations and probabilistic runs.

Theorem 5.2. Let ν be a non-leaking normalised continuous valuation on a coherent ω -algebraic domain D. Then there is a unique probability measure μ on S such that for every compact element x, $\mu(K(x)) = \nu(\uparrow x)$.

Let μ be a probability measure on S. Then the function ν defined on open sets by $\nu(O) = \mu(O \cap \Omega(D))$ is a non-leaking normalised continuous valuation.

According to the result above, probabilistic event structures over a common event structure \mathcal{E} correspond precisely to the probabilistic runs of \mathcal{E} . Among these we can characterise probabilistic event structures *with independence* in terms of the standard measure-theoretic notion of independence. In fact, for such a probabilistic event structure, every two compatible configurations are probabilistically independent, given the common past:

Proposition 5.3. Let v be a configuration valuation on a confusion-free event structure \mathcal{E} . Let μ_v be the corresponding measure as of Propositions 4.1 and Theorem 5.2. Then, v is a configuration valuation with independence iff for every two finite compatible configurations x, y

$$\mu_{v}\Big(K(x)\cap K(y)\mid K(x\cap y)\Big)=\mu_{v}\Big(K(x)\mid K(x\cap y)\Big)\cdot\mu_{v}\Big(K(y)\mid K(x\cap y)\Big).$$

Note that the definition of probabilistic run of an event structure does not require that the event structure is confusion-free. It thus suggests a general definition of a probabilistic event structure as an event structure with a probability measure μ on its maximal configurations, even when the event structure is not confusion-free. This definition, in itself, is however not very informative and we look to an explanation in terms of finite probabilistic runs.

5.2 Finite Runs

What is a finite probabilistic run? Following the analogy heading this section, we want it to be a probability distribution over finite configurations. But which sets are suitable to be the support of such distribution? In interleaving models, the sets of runs of the same length do the job. For event structures this won't do.

To see why consider the event structure with only two concurrent events a, b. The only maximal run assigns probability 1 to the maximal configuration $\{a, b\}$. This corresponds to a configuration valuation which assigns 1 to both $\{a\}$ and $\{b\}$. Now these are two configurations of the same size, but their common "probability" is equal to 2! The reason is that the two configurations are compatible: they do not represent *alternative* choices. We therefore need to represent alternative choices, and we need to represent them all. This leads us to the following definition.

Definition 5.4. Let \mathcal{E} be an event structure. A partial test of \mathcal{E} is a set C of pairwise incompatible configurations of \mathcal{E} . A test is a maximal partial test. A test is finitary if all its elements are finite.

Maximality of a partial test *C* can be characterised equivalently as *completeness*: for every maximal configuration z, there exists $x \in C$ such that $x \subseteq z$. The set of tests, endowed with the Egli-Milner order has an interesting structure: the set of all tests is a complete lattice, while finitary tests form a lattice.

Tests were designed to support probability distributions. So given a sensible valuation on finite configurations we expect it to restrict to probability distributions on tests.

Definition 5.5. Let v be a function $\mathcal{L}_{fin}(\mathcal{E}) \to [0,1]$. Then v is called a test valuation if for all finitary tests C we have v[C] = 1.

Theorem 5.6. Let μ be a probabilistic run of \mathcal{E} . Define $v : \mathcal{L}_{fin}(\mathcal{E}) \to [0,1]$ by $v(x) = \mu(K(x))$. Then v is a test valuation.

Note that Theorem 5.6 is for general event structures. We unfortunately do not have a converse in general. However, there is a converse when the event structure is confusion-free:

Theorem 5.7. Let \mathcal{E} be a confusion-free event structure. Let v be a function $\mathcal{L}_{fin}(\mathcal{E}) \to [0,1]$. Then v is a configuration valuation if and only if it is a test valuation.

The proof of this theorem hinges on a property of tests. The property is that of whether partial tests can be completed. Clearly every partial test can be completed to a test (by Zorn's lemma), but there exist finitary partial tests that cannot be completed to *finitary* tests.

Definition 5.8. A finitary partial test is honest if it is part of a finitary test. A finite configuration is honest if it is honest as partial test.

Proposition 5.9. If \mathcal{E} is a confusion-free event structure and if x is a finite configuration of \mathcal{E} , then x is honest in $\mathcal{L}(\mathcal{E})$.

So confusion-free event structures behave well with respect to honesty. For general event structures, the following is the best we can do at present:

Theorem 5.10. Let v be a test valuation on \mathcal{E} . Let \mathcal{H} be the σ -algebra on $\Omega(\mathcal{L}(\mathcal{E}))$ generated by the shadows of honest finite configurations. Then there exists a unique measure μ on \mathcal{H} such that $\mu(K(x)) = v(x)$ for every honest finite configuration x.

Theorem 5.11. If all finite configurations are honest, then for every test valuation v there exists a unique continuous valuation v, such that $\nu(\uparrow x) = v(x)$.

But, we do not know whether in all event structures, every finite configuration is honest. We conjecture this to be the case. If so this would entail the general converse to Theorem 5.6 and so characterise probabilistic event structures, allowing confusion, in terms of finitary tests.

6 Morphisms

It is relatively straightforward to understand event structures with independence. But how can general test valuations on a confusion-free event structures arise? More generally how do we get runs of arbitrary event structures? We explore one answer in this section. We show how to obtain test valuations as "projections" along a morphism from a configuration valuation with independence on a confusion-free event structure. The use of morphisms shows us how general valuations are obtained through the hiding and renaming of events.

Definition 6.1 ([Win82,WN95]). Given two event structures $\mathcal{E}, \mathcal{E}'$, a morphism $f : \mathcal{E} \to \mathcal{E}'$ is a partial function $f : \mathcal{E} \to \mathcal{E}'$ such that

- whenever $x \in \mathcal{L}(\mathcal{E})$ then $f(x) \in \mathcal{L}(\mathcal{E}')$;
- for every $x \in \mathcal{L}(\mathcal{E})$, for all $e_1, e_2 \in x$ if $f(e_1), f(e_2)$ are both defined and $f(e_1) = f(e_2)$, then $e_1 = e_2$.

A morphism $f : \mathcal{E} \to \mathcal{E}'$ expresses how the occurrence of an event in \mathcal{E} induces a synchronised occurrence of an event in \mathcal{E}' . Some events in \mathcal{E} are hidden (if f is not defined on them) and conflicting events in \mathcal{E} may synchronise with the same event in \mathcal{E}' (if they are identified by f).

The second condition in the definition guarantees that morphisms of event structures "reflect" reflexive conflict $(\# \cup Id_E)$. We now introduce morphisms that reflect tests; such morphisms enable us to define a test valuation on \mathcal{E}' from a test valuation on \mathcal{E} . To do so we need some preliminary definitions. Given a morphism $f: \mathcal{E} \to \mathcal{E}'$, we say that an event of \mathcal{E} is *f*-invisible, if it is not in the domain of *f*. Given a configuration *x* of \mathcal{E} we define x_f to be *x* minus all its maximal *f*-invisible events. Clearly x_f is still a configuration and $f(x) = f(x_f)$. If $x = x_f$, we say that *x* is *f*-minimal.

Definition 6.2. A morphism of event structures $f : \mathcal{E} \to \mathcal{E}'$ is tight when

- if y = f(x) and if $y' \supseteq y$, there exists $x' \supseteq x_f$ such that y' = f(x');
- if y = f(x) and if $y' \subseteq y$, there exists $x' \subseteq x_f$ such that y' = f(x');
- all maximal configurations are *f*-minimal (no maximal event is *f*-invisible).

Proposition 6.3. A tight morphism of event structures is surjective on configurations. Given $f : \mathcal{E} \to \mathcal{E}'$ tight, if C' is a finitary test of \mathcal{E}' then the set of f-minimal inverse images of C' along f is a finitary test in \mathcal{E} .

We now study the relation between valuations and morphisms. Given a function $v : \mathcal{L}_{\text{fin}}(\mathcal{E}) \to [0, +\infty]$ and a morphism $f : \mathcal{E} \to \mathcal{E}'$ we define a function $f(v) : \mathcal{L}_{\text{fin}}(\mathcal{E}') \to [0, +\infty]$ by $f(v)(y) = \sum \{v(x) \mid f(x) = y \text{ and } x \text{ is } f\text{-minimal}\}.$

Proposition 6.4. Let $\mathcal{E}, \mathcal{E}'$ be event structures, v be a test valuation on \mathcal{E} , and $f: \mathcal{E} \to \mathcal{E}'$ a tight morphism. Then the function f(v) is a test valuation on \mathcal{E}' .

Therefore we can obtain a run of a general event structure by projecting a run of a probabilistic event structure with independence. Presently we don't know whether every run can be generated in this way.

7 Related and Future Work

In his PhD thesis, Katoen [Kat96] defines a notion of probabilistic event structure which includes our probabilistic event structures with independence. But his concerns are more directly tuned to a specific process algebra. So in one sense his work is more general—his event structures also possess nondeterminism—while in another it is much more specific in that it does not look beyond local probability distributions at cells. Völzer [Voe01] introduces similar concepts based on Petri nets and a special case of Theorem 5.10. Benveniste et al. have an alternative definition of probabilistic Petri nets in [BFH03], and there is clearly an overlap of concerns though some significant differences which require study. We have explored how to add probability to the independence model of event structures. In the confusion-free case, this can be done in several equivalent ways: as valuations on configurations; as continuous valuations on the domain of configurations; as probabilistic runs (probability measures over maximal configurations); and in the simplest case, with independence, as probability distributions existing locally and independently at cells. Work remains to be done on a more operational understanding, in particular on how to understand probability adjoined to event structures which are not confusion-free. This involves relating probabilistic event structures to interleaving models like Probabilistic Automata [Seg95] and Labelled Markov Processes [DEP02].

Acknowledgments

The first author wants to thank Mogens Nielsen, Philippe Darondeau, Samy Abbes and an anonymous referee.

References

[AJ94]	S. Abramsky and A. Jung. Domain theory. In <i>Handbook of Logic in Com-</i> <i>puter Science</i> , volume 3, Clarendon Press, 1994
[Alv00]	M. Alvarez-Manilla. <i>Measure Theoretic Results for Continuous Valuations</i> on Partially Ordered Spaces. PhD thesis, University of London - Imperial
	College of Science, Technology and Medicine, 2000.
[AES00]	M. Alvarez-Manilla, A. Edalat, and N. Saheb-Djaromi. An extension result
	for continuous valuations. Journal of the London Mathematical Society, 61(2):629–640, 2000.
[BFH03]	A. Benveniste, E. Fabre, and S. Haar. Markov nets: Probabilistic models
	for distributed and concurrent systems. <i>IEEE Transactions on Automatic Control</i> , 48(11):1936–1950, 2003.
[dAHJ01]	L. de Alfaro, T. A. Henzinger, and R. Jhala. Compositional methods for probabilistic systems. In <i>Proc. 12th CONCUR</i> , volume 2154 of <i>LNCS</i> , pages
	351–365, 2001.
[DEP02]	J. Desharnais, A. Edalat, and P. Panangaden. Bisimulation for labelled markov processes. <i>Information and Computation</i> , 179(2):163–193, 2002.
[Eda95]	A. Edalat. Domain theory and integration. <i>Theoretical Computer Science</i> , 151(1):163–193. 13 November 1995.
LIP891	C. Jones and G. D. Plotkin. A probabilistic powerdomain of evaluations.
[31 07]	In Proceedings of 4th LICS, pages 186–195, 1989.
[Kat96]	JP. Katoen. <i>Quantitative and Qualitative Extensions of Event Structures</i> . PhD thesis, University of Twente, 1996.
[KP93]	G. Kahn and G. D. Plotkin. Concrete domains. <i>Theoretical Computer</i> Science, 121(1-2):187–277, 1993
[LS91]	K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. Information and Computation 94(1):1–28 1991
[Mi189]	R Milner Communication and Concurrency Prentice Hall 1989
[NPW81]	M. Nielsen, G. D. Plotkin, and G. Winskel. Petri nets, event structures and domains, part I. <i>Theoretical Computer Science</i> , 13(1):85–108, 1981.

- [Seg95] R. Segala. Modeling and Verification of Randomized Distributed Real-Time Systems. PhD thesis, MIT, 1995.
- [Voe01] H. Völzer. Randomized non-sequential processes. In Proceedings of 12th CONCUR, volume 2154 of LNCS, pages 184–201, 2001. Extended version as Technical Report 02-28 - SVRC - University of Queensland.
- [Var03] D. Varacca. Probability, nondeterminism and Concurrency. Two denotational models for probabilistic computation. PhD thesis, BRICS - Aarhus University, 2003. Available at http://www.brics.dk/~varacca.
- [VVW04] D. Varacca, H. Völzer and G. Winskel. Probabilistic Event Structures and Domains. BRICS Technical Report RS-04-10 - Aarhus University, 2004.
- [Win82] G. Winskel. Event structure semantics for CCS and related languages. In Proceedings of 9th ICALP, volume 140 of LNCS, pages 561–576. Springer, 1982.
- [Win87] G. Winskel. Event structures. In Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course, Bad Honnef, September 1986, volume 255 of LNCS, pages 325–392. Springer, 1987.
- [WN95] G. Winskel and M. Nielsen. Models for concurrency. In *Handbook of logic in Computer Science*, volume 4. Clarendon Press, 1995.

Appendix: Domain Theory—Basic Notions

We briefly recall some basic notions of domain theory (see e.g. [AJ94]). A *directed complete partial order* (*DCPO*) is a partial order where every directed set Y has a least upper bound $\bigsqcup Y$. An element x of a DCPO D is *compact* (or *finite*) if for every directed Y and every $x \leq \bigsqcup Y$ there exists $y \in Y$ such that $x \leq y$. The set of compact elements is denoted by Cp(D). A DCPO is an *algebraic domain* if or every $x \in D$, x is the directed least upper bound of $\downarrow x \cap Cp(D)$. It is ω -algebraic if Cp(D) is countable.

In a partial order, two elements are said to be *compatible* if they have a common upper bound. A subset of a partial order is *consistent* if every two of its elements are compatible. A partial order is *coherent* if every consistent set has a least upper bound.

The *Egli-Milner* order on subsets of a partial order is defined by $X \leq Y$ if for all $x \in X$ there exists $y \in Y$, $x \leq y$ and for all $y \in Y$ there exists $x \in X$, $x \leq y$. A subset X of a DCPO is *Scott open* if it is upward closed and if for every directed set Y whose least upper bound is in X, then $Y \cap X \neq \emptyset$. Scott open sets form the *Scott topology*.

Session Types for Functional Multithreading

Vasco Vasconcelos¹, António Ravara², and Simon Gay³

 ¹ Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa, 1749-016 Lisboa, Portugal vv@di.fc.ul.pt
 ² Departamento de Matemática, Instituto Superior Técnico, 1049-001 Lisboa, Portugal amar@math.ist.utl.pt

³ Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK simon@dcs.gla.ac.uk

Abstract. We define a language whose type system, incorporating session types, allows complex protocols to be specified by types and verified by static typechecking. A session type, associated with a communication channel, specifies the state transitions of a protocol and also the data types of messages associated with transitions; thus typechecking can verify both correctness of individual messages and correctness of sequences of transitions. Previously session types have mainly been studied in the context of the π -calculus; instead, our formulation is based on a multi-threaded functional language with side-effecting input/output operations. Our typing judgements statically describe dynamic changes in the types of channels, our channel types statically track aliasing, and our function types not only specify argument and result types but also describe changes in channels. We formalize the syntax, semantics and typing rules of our language, and prove subject reduction and runtime type safety theorems.

Keywords: Session types, static typechecking, concurrent programming, specification of communication protocols.

1 Introduction

Communication in distributed systems is typically structured around protocols, which specify the sequence and form of messages passing over communication channels. Correctness of such systems implies that protocols are obeyed.

The theory of *session types* [9,10,18,5] allows the specification of a protocol to be expressed as a type; when a communication channel is created, a session type is associated with it. Such a type specifies not only the data types of individual messages, but also the state transitions of the protocol and hence the allowable sequences of messages. By extending the standard methodology of static typechecking, it becomes possible to verify, at compile-time, that an agent using the channel does so in accordance with the protocol.

The theory of session types has been developed in the context of the π -calculus [13,17], an idealized concurrent programming language which focuses

on inter-process communication. Session types have not yet been incorporated into a mainstream programming language, or even studied theoretically in the context of a standard language paradigm: functional, imperative or object-oriented. Vallecillo *et al.* [19] use session types to add behavioural information to the interfaces of CORBA objects, and use Gay and Hole's [5] theory of subtyping to formalize compatibility and substitutability of components, but they have not attempted to design a complete language.

The Vault [2] and Cyclone [8] languages extend C with facilities for safe control of stateful resources. In Cyclone, locks must be acquired and released; Vault goes further by allowing operations on a resource to be statically checked against an automaton which specifies valid transitions. In contrast, session types are specialized to communication channels as a particular kind of resource, but as a result they enable further typechecking in association with each state transition: typechecking verifies the types of individual messages, as well as verifying that a sequence of messages obeys a given protocol. (These languages are further discussed in section 7.)

In previous work [4] we have presented a language supporting typed functional programming with inter-process communication channels, but we only considered individual processes in isolation. Here we address collections of functional threads communicating via channels. This formulation allows us to prove a runtime safety property: well-typed programs do not misuse channels.

By transferring the concept of session types from the π -calculus to a multithreaded functional language with side-effecting input/output operations, we show that static checking of session types could be added to a language such as Concurrent ML [16], at least without imperative features. In particular we have addressed the key differences between a conventional programming style and the programming notation of the π -calculus:

- The operations on channels are independent terms, rather than prefixes of processes, so we have introduced a new form of typing judgement which describes the effect of a term on channel environment.
- We have separated creation and naming of channels, and because this introduces the possibility of aliasing, we represent the types of channels by indirection from the main type environment to the channel environment.

The structure of the paper is as follows. In Section 2 we explain session types in connection with a progressively more sophisticated example. Sections 3, 4 and 5 define the syntax, operational semantics and type system of our language. In Section 6 we present the runtime safety result. In Sections 7 and 8 we discuss related and future work.

2 Session Types and the Maths Server

Input, Output and Sequencing Types. First consider a server which provides a single operation: addition of integers. A suitable protocol can be defined as follows.

The client sends two integers. The server sends an integer which is their sum, then closes the connection.

The corresponding session type, from the server's point of view, is

S = ?Int.?Int.!Int.End

in which ? means *receive*, ! means *send*, dot (.) is *sequencing*, and End indicates the end of the session. The type does not correspond precisely to the specification, because it does not state that the server calculates the sum. However, the type captures the parts of the specification which we can reasonably expect to verify statically. The server communicates with a client on a channel called u; we think of the client engaging in a *session* with the server, using the channel u for communication. In our language, the server looks like this:

server u =let x = receive u in let y = receive u in send x + y on u

or more concisely: send (receive u) + (receive u) on u.

Interchanging ? and ! yields the type describing the client side of the protocol:

$\overline{S} = !!nt.!!nt.?!nt.End$

and a client implementation uses the server to add two particular integers; the *code* may use x but cannot use the channel u except for closing it.

```
client u = \text{send } 2 on u
send 3 on u
let x = \text{receive } u in code
```

Branching Types. Now let us modify the protocol and add a negation operation to the server.

The client selects one of two commands: *add* or *neg*. In the case of *add* the client then sends two integers and the server replies with an integer which is their sum. In the case of *neg* the client then sends an integer and the server replies with an integer which is its negation. In either case, the server then closes the connection.

The corresponding session type, for the server side, uses the constructor & (*branch*) to indicate that a choice is offered.

```
S = \& \langle add: ? \text{Int.?Int.!Int.End}, neg: ? \text{Int.!Int.End} \rangle
```

Both services must be implemented. We introduce a case construct:

```
server u = case u of {

add \Rightarrow send (receive u) + (receive u) on u

neg \Rightarrow send - (receive u) on u }
```

The type of the client side uses the dual constructor \oplus (*choice*) to indicate that a choice is made.

 $\overline{S} = \bigoplus \langle add : ! \text{Int.!Int.?Int.End}, neg : ! \text{Int.?Int.End} \rangle$

A client implementation makes a particular choice, for example:

$addClient \ u = select \ add \ on \ u$	$negClient \ u = {\sf select} \ neg \ {\sf on} \ u$		
send 2 on u	send 7 on u		
send 3 on u	let $x =$ receive u in $code$		
let $x =$ receive u in $code$			

Note that the type of the subsequent interaction depends on the label which is selected. In order for typechecking to be decidable, it is essential that the label *add* or *neg* appears as a literal name in the program; labels cannot result from computations.

If we add a square root operation, *sqrt*, then as well as specifying that the argument and result have type **Real**, we must allow for the possibility of an error (resulting in the end of the session) if the client asks for the square root of a negative number. This is done by using the \oplus constructor on the server side, with options *ok* and *error*. The complete English description of the protocol is starting to become lengthy, so we will omit it and simply show the type of the server side.

$$\begin{split} S &= \& \langle add: ? | \texttt{nt.?} \texttt{Int.!} \texttt{Int.End}, \\ neg: ? | \texttt{nt.!} \texttt{Int.End}, \\ sqrt: ? \texttt{Real} . \oplus \langle ok: ! \texttt{Real.End}, error: \texttt{End} \rangle \rangle \end{split}$$

This example shows that session types allow the description of protocols that cannot be easily accommodated with objects, that is, with sequences of the form: select a method; send the arguments; receive the result.

Recursive Types. A more realistic server would allow a session to consist of a sequence of commands and responses. The corresponding type must be defined recursively, and it is useful to include a *quit* command. Here is the type of the server side:

$$\begin{split} S &= \&\langle add : ? \texttt{Int.?}\texttt{Int.!}\texttt{Int.}S, \\ neg : ? \texttt{Int.!}\texttt{Int.}S, \\ sqrt : ? \texttt{Real}. \oplus \langle ok : !\texttt{Real}.S, error : S \rangle, \\ quit : \texttt{End} \rangle \end{split}$$

The server is now implemented by a recursive function, in which the positions of the recursive calls correspond to the recursive occurrences of S in the type definition. To simplify the theory we decided not to include recursive types in this paper; the interested reader may refer to report [4].

Function Types. We have not mentioned the type of the *server* itself. Clearly, it accepts a channel (in state &: $\langle add : ..., neg : ... \rangle$), and returns nothing (described by the Unit type). The body of the function "consumes" the channel, leaving it in a state ready to be closed (described by type End). We write all this as follows, where *c* is the (runtime) channel denoted by the (program) variable *u*.

```
server :: c: & \langle add: ..., neg: ... \rangle; Chan c \rightarrow \text{Unit}; c: End
server u = \text{case } u of \{add \Rightarrow ..., neg \Rightarrow ... \}
```

Note how the function type describes, not only the type of the parameter and that of the result, but also, its effect on channel c. It can also be useful to send functions on channels. For example we could add the component¹

 $eval: ?(Int \rightarrow Bool).?Int.!Bool.End$

to the branch type of the *server*, with corresponding server code, to be placed within the server's **case** above.

$$eval \Rightarrow$$
 send (receive u)(receive u) on u

A client which requires a primality test service (perhaps the server has fast hardware) can be written as follows.

 $primeClient :: c: \oplus \langle add : ..., neg : ..., eval : ... \rangle$; Chan $c \to Unit; c: End$ primeClient u = select eval on u send isPrime on u send bigNumber on ulet x = receive u in code

Establishing a Connection. How do the client and the server reach a state in which they both know about channel c? We follow Takeuchi, Kubo and Honda [18], and propose a pair of constructs: request v for use by clients, and accept v for use by servers. In use, request and accept occur in separate threads, and interact with each other to create a new channel. The value v in both request and accept, denotes the common knowledge of the two threads: a *shared name* used solely for the creation of new channels. We may then write:

```
server :: [\&\langle add: \dots, neg: \dots, eval: \dots \rangle] \to \text{Unit}
server \ x = \text{let} \ u = \text{accept} \ x \text{ in } (\text{case } u \text{ of } \dots; \text{close } u)
negClient :: [\&\langle add: \dots, neg: \dots, eval: \dots \rangle] \to \text{Unit}
negClient \ x = \text{let} \ u = \text{request} \ x \text{ in } (\text{select } neg \text{ on } u \dots; \text{close } u)
```

Note that the same type for the shared name x is used both for the server and for the client; it is the accept/request construct that distinguishes one from

¹ We often omit the empty channel environment on each side of the arrow.

the other. This is also where we introduce the operation to close a channel: accept/request creates a channel; close destroys it.

Sharing Names. In order for a name to become known by a client and a server, it must be created somewhere and distributed to both. To create a new, potentially shared, name, we write **new**. To distribute it to a second thread, we fork a new thread, in whose code the name occurs.² Our complete system creates a name x and launches three threads (a server and two clients), all sharing the newly created name.

system :: Unit system = let x = new infork negClient x; fork addClient x; fork server x

Given the above implementation of *server*, one of the clients will be forever requesting x. Fortunately, it is easy to extend the *server* to accept more than one connection in its life time.

server :: $[\&\langle add: \ldots, neg: \ldots, eval: \ldots \rangle] \rightarrow Unit$ server $x = let \ u = accept \ x \text{ in fork } (case \ u \text{ of } \ldots; close \ u)$ server x

Sending Channels on Channels. Imagine two clients that need to cooperate in their interaction with the server: one client establishes a connection, selects the *neg* operation, and sends the argument; the second client receives the result. After **selecting** *neg*, the first client must provide the second with the channel to the server. In order to do so, both clients must share a name of type ?(?Int.End).End (call this type *S*) and establish a connection for the sole purpose of transmitting the server channel.

$\mathit{askNeg} :: [\langle \mathit{add} \colon \ldots angle] o \ [S] o Unit$	$\mathit{getNeg}::[S] ightarrow Unit$
$askNeg \ x \ y = $ let $u =$ request x in	$getNeg \ y = let \ w = accept \ y \ in$
select neg on u ; send 7 on u	let $u =$ receive w in
let $w = request y$ in	let $i =$ receive u in
send u on w ; close w	close u ; close w ; $code$

It is instructive to follow the evolution of the state (the type) of channels c and d, connected to variables u and w, respectively. After the execution of the first line of *getNeg*, d has type S = ?(?Int.End).End; after the second line, d is reduced to End, but c shows up with type ?Int.End; after the third line both channels are of type End, that is, ready to be closed. By the end of the fourth line, we gather no more information on channels c and d, for they are now closed. That is the sort of analysis our type system performs.

² Alternatively, we may send \boldsymbol{x} on an existing channel.

After sending a channel, no further interaction on the channel is possible. Note that askNeg cannot close u, for otherwise the channel's client side would be closed twice (in askNeg and in getNeg). On the other hand, channel w must be closed at both its ends, by askNeg and by getNeg.

Channel Aliasing. As soon as we separate creation and naming of channels, aliasing becomes an issue. Consider the function below.

 $sendSend \ u \ v = send \ 1 \text{ on } u; send \ 2 \text{ on } v$

Function *sendSend* can be used in a number of different ways including the one where u and v become aliases for a single underlying channel.

 $sendTwice :: c: !Int.!Int.End; Chan c \rightarrow Unit; c: End$ sendTwice w = sendSend w w

Clearly our type system must track aliases in order to be able to correctly typecheck programs such as this. Our approach is to introduce indirection into type environments. In the body of function *sendSend*, the types of u and v are both Chan c. The state of c, initially !Int.!Int.End, is recorded separately.

Free Variables in Functions. If we write

$$sendFree \ v = send \ 1 \ on \ u; send \ 2 \ on \ v$$

then function sendSend becomes $\lambda u.sendFree$. In order to type sendTwice, thus effectively aliasing u and v in sendSend, we must have³

 $sendFree :: c: !Int.!Int.End; Chan c \rightarrow Unit; c: End$ $sendSend :: c: !Int.!Int.End; Chan c \rightarrow Chan c \rightarrow Unit; c: End$

in a typing environment associating the type Chan c to the free variable u of *sendFree*. However, if aliasing u and v is not sought, then we must have

 $sendFree :: c: !Int.End, d: !Int.End; Chan <math>c \rightarrow Unit; c: End, d: End$ $sendSend :: c: !Int.End, d: !Int.End; Chan <math>c \rightarrow Chan d \rightarrow Unit; c: End, d: End$

in a typing environment containing u: Chan d. Note how this type for *sendFree* captures channel changes, parameters to the function or not.

Polymorphism. We have seen that *sendFree* admits at least two different types. In order to allow for code reuse we work with a type-free syntax, and type our functions as many times as needed, potentially with different types. The paragraph above showed a *share/not-share* kind of polymorphism. Other forms include *channel polymorphism* and *session polymorphism*. For the former consider

sendTwiceSendTwice :: c: S, d: S; Chan $c \rightarrow$ Chan $d \rightarrow$ Unit; c: End, d: End sendTwiceSendTwice x y = sendTwice x; sendTwice y

³ We abbreviate $\Sigma; T \to (\Sigma; U \to V; \Sigma'); \Sigma'$ to $\Sigma; T \to U \to V; \Sigma'$.

 $\begin{array}{l} v ::= c \mid n \mid x \mid \lambda x.e \mid \operatorname{rec} x.v \mid \operatorname{true} \mid \operatorname{false} \mid \operatorname{unit} \\ e ::= t \mid vv \mid \operatorname{if} v \operatorname{then} e \operatorname{else} e \mid \operatorname{new} \mid \operatorname{accept} v \mid \operatorname{request} v \mid \\ & \operatorname{send} v \operatorname{on} v \mid \operatorname{receive} v \mid \operatorname{case} v \operatorname{of} \{l_i \Rightarrow e_i\}_{i \in I} \mid \operatorname{select} l \operatorname{on} v \mid \operatorname{close} v \\ t ::= v \mid \operatorname{let} x = e \operatorname{in} t \mid \operatorname{fork} e; t \\ C ::= \langle t \rangle \mid (C \mid C) \mid (\nu n)C \mid (\nu c)C \end{array}$



 $(C, |, \langle \text{unit} \rangle)$ is a commutative monoid $(\nu n)C_1 | C_2 \equiv (\nu n)(C_1 | C_2)$ if n not free in C_2 $(\nu c)C_1 | C_2 \equiv (\nu c)(C_1 | C_2)$ if c not free in C_2

Fig. 2. Structural congruence

where *S* is !Int.!Int.End.Here *sendTwice* must be typed once with channel *c*, and another with channel *d*. For the latter we have:

 $sendQuad :: c: !Int.!Int.!Int.End; Chan c \rightarrow Unit; c: End$ sendQuad x = sendTwice x; sendTwice x

where *sendTwice* must be typed once with *c*: !Int.!Int.!Int.end, and a second time with *c*:!Int.!Int.End.

3 Syntax

Most of the syntax of our language has been illustrated in the previous section; here we define it formally by the grammar in Figure 1.

We use channel identifiers c, ..., name identifiers n, ..., term variables x, ...,and labels l, ..., and define values v, terms e, threads t, and configurations C. Channel identifiers and name identifiers are not available in the top-level syntax of threads; they arise only during reduction, in a request/accept synchronization, and in a new operation, respectively, as described in section 4.

In section 2 we used several derived constructors. An expression e; t (sometimes implied in our examples by the indentation) is an abbreviation for let y = e in t, provided y does not occur free in t. Idioms like send (receive c) (receive c) on c need appropriate de-sugaring into consecutive lets, making the evaluation order explicit. We sometimes "terminate" threads with an expression rather than a value: a thread e is short for let $x = e \ln x$. Recursive function definitions must be made explicit with rec.

4 Operational Semantics

The binding occurrences are x in $\lambda x.e$, rec x.e, let x = e in t, n in (vn)C and c in (vc)C. Free and bound identifiers are defined as usual and we work up

$$\begin{array}{l} \langle \operatorname{let} x = \operatorname{request} n \operatorname{in} t_1 \rangle \mid \langle \operatorname{let} y = \operatorname{accept} n \operatorname{in} t_2 \rangle \rightarrow \\ (\nu c)(\langle \operatorname{let} x = c \operatorname{in} t_1 \rangle \mid \langle \operatorname{let} y = c \operatorname{in} t_2 \rangle) & (\operatorname{R-INIT}) \\ \langle \operatorname{let} x = \operatorname{receive} c \operatorname{in} t_1 \rangle \mid \langle \operatorname{let} y = \operatorname{send} v \text{ on } c \operatorname{in} t_2 \rangle \rightarrow \langle \operatorname{let} x = v \operatorname{in} t_1 \rangle \mid \langle \operatorname{let} y = \operatorname{unit} \operatorname{in} t_2 \rangle \\ (\operatorname{R-COM}) \\ \langle \operatorname{let} x = \operatorname{case} c \operatorname{of} \{l_i \Rightarrow e_i\} \operatorname{in} t_1 \rangle \mid \langle \operatorname{let} y = \operatorname{select} l_j \operatorname{on} c \operatorname{in} t_2 \rangle \rightarrow \\ \langle \operatorname{let} x = \operatorname{case} c \operatorname{of} \{l_i \Rightarrow e_i\} \operatorname{in} t_1 \rangle \mid \langle \operatorname{let} y = \operatorname{unit} \operatorname{in} t_2 \rangle \\ (\operatorname{R-COM}) \\ \langle \operatorname{let} x = \operatorname{close} c \operatorname{in} t_1 \rangle \mid \langle \operatorname{let} y = \operatorname{close} c \operatorname{in} t_2 \rangle \rightarrow \\ \langle \operatorname{let} x = \operatorname{close} c \operatorname{in} t_1 \rangle \mid \langle \operatorname{let} y = \operatorname{close} c \operatorname{in} t_2 \rangle \rightarrow \\ \langle \operatorname{let} x = \operatorname{close} c \operatorname{in} t_1 \rangle \mid \langle \operatorname{let} y = \operatorname{close} c \operatorname{in} t_2 \rangle \rightarrow \langle \operatorname{let} x = \operatorname{unit} \operatorname{in} t_1 \rangle \mid \langle \operatorname{let} y = \operatorname{unit} \operatorname{in} t_2 \rangle \\ (\operatorname{R-CLOSE}) \\ \langle \operatorname{let} x = \operatorname{new} \operatorname{in} t \rangle \rightarrow \langle \operatorname{vn} \rangle \langle \operatorname{let} x = n \operatorname{in} t \rangle \\ \langle \operatorname{R-NEW} \rangle \\ \langle \operatorname{fork} e; t \rangle \rightarrow \langle t \rangle \mid \langle \operatorname{let} x = e \operatorname{in} \operatorname{unit} \rangle \\ \langle \operatorname{R-NEW} \rangle \\ \langle \operatorname{let} x = \operatorname{if} \operatorname{false} \operatorname{then} e \operatorname{else} e' \operatorname{in} t \rangle \rightarrow \langle \operatorname{let} x = e \operatorname{in} t \rangle \\ \langle \operatorname{R-FORK} \rangle \\ \langle \operatorname{let} x = \operatorname{if} \operatorname{false} \operatorname{then} e \operatorname{else} e' \operatorname{in} t \rangle \rightarrow \langle \operatorname{let} x = e \operatorname{in} t \rangle \\ \langle \operatorname{R-IFT} \rangle \\ \langle \operatorname{let} x = \operatorname{if} \operatorname{false} \operatorname{then} e \operatorname{else} e' \operatorname{in} t \rangle \rightarrow \langle \operatorname{let} x = e' \operatorname{in} t \rangle \\ \langle \operatorname{R-IFF} \rangle \\ \langle \operatorname{let} x = (\operatorname{c} (\lambda y. e) v \operatorname{in} t \rangle \rightarrow \langle \operatorname{let} x = (\operatorname{let} y = \operatorname{vin} e) \operatorname{in} t \rangle \\ \langle \operatorname{R-APP} \rangle \\ \langle \operatorname{let} x = (\operatorname{rec} y. v) u \operatorname{in} t \rangle \rightarrow \langle \operatorname{let} x = (\operatorname{let} y = \operatorname{vin} v \operatorname{uv}) \operatorname{in} t \rangle \\ \langle \operatorname{R-REC} \rangle \\ \langle \operatorname{let} x = v \operatorname{in} t \rangle \rightarrow \langle \operatorname{let} x = e \operatorname{in} v \rangle \\ \langle \operatorname{R-BETA} \rangle \\ \langle \operatorname{R-BETA} \rangle \\ \langle \operatorname{R-BETA} \rangle \\ \langle \operatorname{R-DeT} \rangle \\ \langle \operatorname{R-COF} \rangle \\ \langle \operatorname{R-COF} \rangle \langle \operatorname{R-CO} \rangle \langle \operatorname{R-C} \rangle \rangle \langle \operatorname{R-CO} \rangle \langle \operatorname{R-C} \rangle \rangle \\ \langle \operatorname{R-COF} \rangle \langle \operatorname{R-COF$$

In R-INIT, c is not free in t_1, t_2 ; in R-NEW, n is not free in t.

Fig. 3. Reduction rules

to α -equivalence. Substitution, of values for variables, is defined as expected. We define a reduction semantics on configurations (figure 3), making use of a simple structural congruence relation [13] (figure 2), allowing rearrangement of the threads in a configuration, so that reduction may happen.⁴

We now explain the reduction rules. R-INIT synchronizes two threads on a *shared name z*, creating a new channel *c* known to both threads. Rules R-COM, R-BRANCH, and R-CLOSE synchronize two threads on a *channel c*: R-COM transmits a value v from one thread to the other; R-BRANCH, rather than transmitting a value, chooses one of the branches in the **case** thread; and R-CLOSE closes a channel in *both* threads simultaneously. R-NEW creates a new name n, and records the fact that the name is potentially shared, by means of a (vn) in the resulting configuration. The last four rules allow reduction to happen underneath restriction, parallel composition, and structural congruence.

Unlike other thread models, the value a thread reduces to is not communicated back to its parent thread (the one that forked the terminating thread).

⁴ We could easily arrange for structural congruence to garbage collect all threads of the form $\langle v \rangle$, for v closed.

$$\begin{split} S &::= ?D.S \mid !D.S \mid ?S.S \mid !S.S \mid \&\langle l_i \colon S_i \rangle_{i \in I} \mid \oplus \langle l_i \colon S_i \rangle_{i \in I} \mid \mathsf{End} \mid \bot \\ \Sigma &::= \emptyset \mid \Sigma, c \colon S \quad (c \colon S' \text{ not in } \Sigma) \\ D &::= \mathsf{Bool} \mid \mathsf{Unit} \mid \Sigma; T \to T; \Sigma \mid [S] \\ T &::= D \mid \mathsf{Chan} \ c \end{split}$$

Fig. 4. Syntax of types

Such behaviour would have to be programmed by arranging for both threads to share a channel and explicitly **sending** the result back to the parent.

5 Typing

The syntax of types is described in figure 4. We define session types S, channel environments Σ , data types D, and term types T. The type Chan c represents the type of the channel with identity c; the session type associated with c is recorded separately in a channel environment Σ . Channel type bottom, \bot , denotes a channel that has been closed or that is already in use by two threads, hence that cannot be used further. Similarly to channel and name identifiers, \bot is not available at the top level syntax, arising only via the channel environment composition operator, $\Sigma_1 \bullet \Sigma_2$, defined below. Among datatypes we have channel-state annotated functional types $\Sigma; T \to T; \Sigma$, and types for names [S] capable of establishing sessions of type S.

The type system is presented in figures 5, 6, and 7. Typing judgements for configurations are of the form $\Gamma \vdash \Sigma \triangleright C \triangleleft \Sigma'$ where Γ is a map from variables and names to types, and Σ, Σ' are channel environments as in section 3. Judgements for expressions $\Gamma \vdash \Sigma \triangleright e: T \triangleleft \Sigma'$ also describe the type of the expression, and those for constants $\Gamma \vdash v: T$ do not mention channel environments, for constants, having no behaviour, do not change channels. The difference between Σ and Σ' reflects the effect of an expression (or a configuration) on the types of channels, for example

x: Chan $c \vdash c$: ?Int.End \triangleright receive x: Int $\triangleleft c$: End

Typing Values (Figure 5). T-CHAN that says that a channel named c has type Chan c. The actual type (or state) of channel c is to be found in a channel environment Σ , in the rules for expressions. In T-ABS, the initial and final channel environments of the function body are recorded in the function type.

$$\begin{split} \Gamma \vdash \mathsf{true:} \ \mathsf{Bool} & \Gamma \vdash \mathsf{false:} \ \mathsf{Bool} & \Gamma \vdash \mathsf{unit:} \ \mathsf{Unit} & (\mathsf{T}\text{-}\mathsf{CONST}) \\ \Gamma, x \colon T \vdash x \colon T & \Gamma \vdash c \colon \mathsf{Chan} \ c & (\mathsf{T}\text{-}\mathsf{VAR}, \mathsf{T}\text{-}\mathsf{CHAN}) \\ \frac{\Gamma, x \colon T \vdash \Sigma \triangleright e \colon U \triangleleft \Sigma'}{\Gamma \vdash \lambda x.e \colon (\Sigma; T \to U; \Sigma')} & \frac{\Gamma, x \colon T \vdash v \colon T}{\Gamma \vdash \mathsf{rec} \ x.v \colon T} & (\mathsf{T}\text{-}\mathsf{ABS}, \mathsf{T}\text{-}\mathsf{Rec}) \end{split}$$

Fig. 5. Typing rules for values

$$\frac{\Gamma \vdash v: \text{Chan } c}{\Gamma \vdash \Sigma, c: ?D.S \triangleright \text{ receive } v: D \triangleleft \Sigma, c: S}$$
(T-ReceiveD)
$$\Gamma \vdash v: \text{Chan } c$$
(T-ReceiveD)

$$\overline{\Gamma \vdash \Sigma, c: ?S'.S \triangleright \text{ receive } v: \text{ Chan } d \triangleleft (\Sigma \bullet d: S'), c: S}$$

$$(1 \text{-RECEIVES})$$

$$\overline{\Gamma \vdash v: D} = \overline{\Gamma \vdash v': \text{ Chan } c}$$

$$\frac{1 + v \cdot D}{\Gamma + \Sigma, c \colon !D.S \triangleright \text{send } v \text{ on } v' \colon \text{Unit} \triangleleft \Sigma, c \colon S}$$
(T-SENDD)

$$\frac{1 \vdash v: \operatorname{Chan} a}{\Gamma \vdash \Sigma, c: !S'.S, d: S' \triangleright \operatorname{send} v \text{ on } v': \operatorname{Unit} \triangleleft \Sigma, c: S}$$
(T-SENDS)
$$\frac{\Gamma \vdash v: \operatorname{Chan} c}{\Gamma \vdash v: \operatorname{Chan} c} = i \in I$$

$$\frac{T \vdash v: \operatorname{Chan} c \quad j \in I}{\Gamma \vdash \Sigma, c: \oplus \langle l_i: S_i \rangle_{i \in I} \triangleright \operatorname{select} l_j \text{ on } v: \operatorname{Unit} \triangleleft \Sigma, c: S_j}$$
(T-SELECT)

$$\frac{I \vdash v: \operatorname{Chan} c}{I \vdash \Sigma, c: \& \langle l_i: S_i \rangle_{i \in I} \triangleright \operatorname{case} v \text{ of } \{l_i \Rightarrow e_i\}_{i \in I}: T \triangleleft \Sigma'}$$
(T-CASE)

$$\frac{\Gamma \vdash v: \operatorname{Chan} c}{\Gamma \vdash \Sigma, c: \operatorname{End} \triangleright \operatorname{close} v: \operatorname{Unit} \triangleleft \Sigma} \qquad \frac{\Gamma \vdash \Sigma_1 \triangleright e: \lrcorner \triangleleft \Sigma'_1 \qquad \Gamma \vdash \Sigma_2 \triangleright t: T \triangleleft \Sigma'_2}{\Gamma \vdash \Sigma_1 \bullet \Sigma_2 \triangleright \operatorname{fork} e; t: T \triangleleft \Sigma'_1 \bullet \Sigma'_2}$$

$$(T-\operatorname{Close} T-\operatorname{Fork})$$

Fig. 6. Typing rules for expressions

Typing Expressions (Figure 6). There are two rules for receive and two rules for send, for these constructors are overloaded: they allow transmission of data as well as channels. In T-RECEIVED, the prefix ?D., of the type for channel c, is consumed, provided that we are receiving on a value aliased to channel c (of type Chan c). In T-RECEIVES, we receive a channel, that we decided to call d; the type of the expression is Chan d, and we add a new entry to the final channel environment, where we record the type for d. The particular form of the final channel environment allows the continuation to hold both ends of the channel. The rules T-SENDD and T-SENDS, for sending values and channels, are similar. In T-SELECT, the type for c in the final channel environment is that of branch l_i in the type for c in the source channel environment. In T-CASE, all branches must produce the same final channel environment. This enables us to know the environment for any code following the Case, independently of which branch is chosen at runtime. The same applies to the two branches of the conditional in T-IF. Rule T-CLOSE requires that the channel must be ready to be closed (of

type End). We replace the type of c by \perp to mean that no further interaction at c is possible.

Rules T-REQUEST and T-ACCEPT both introduce a new channel c in the channel environment, of *dual polarities* [5,9,10,18,19]. The dual of a session type S, denoted \overline{S} , is defined for all session types except \bot , and is obtained by interchanging output ! and input ?, and by interchanging branching & and selection \oplus , and leaving S otherwise unchanged. In proofs we use an inductive definition.

In T-APP, the initial and final channel environments in the type of the function are released into the typing for the application. T-VAL says that constants do not affect the state of channels. Expression **new** has any type of the form [S], denoting a name that, when shared by two threads, is able to produce (via **accept/request**) new channels of type S.

Rule T-FORK composes the initial and the final channel environments of two configurations, by checking that the types of the channels occurring in both environments are dual. The *composition of two channel environments*, $\Sigma_1 \bullet \Sigma_2$, is defined only when $\Sigma_1(c) = \overline{\Sigma_2(c)}$, for all $c \in \text{dom } \Sigma_1 \cap \text{dom } \Sigma_2$. In this case $\text{dom}(\Sigma_1 \bullet \Sigma_2) = \text{dom } \Sigma_1 \cup \text{dom } \Sigma_2$, and $(\Sigma_1 \bullet \Sigma_2)(c)$ is \perp when $c \in \text{dom } \Sigma_1 \cap$ $\text{dom } \Sigma_2$, and is $\Sigma_i(c)$ when $c \in \text{dom } \Sigma_i \setminus \text{dom } \Sigma_{3-i}$, for i = 1, 2.

Rule T-POLYLET allow types the various forms of polymorphism identified in section 2, by separately typing different copies of the polymorphic value [14].

Typing Configurations (Figure 7). T-PAR is similar to T-FORK. T-NEWN makes sure that only names are used to start sessions. T-NEWC says that a channel must be used with dual modes by exactly two threads; its side condition ensures that channels are closed.

Subject Reduction. Our Subject Reduction theorem describes the evolution of the channel environment as the program is executed. The invariance of Σ' during reduction steps reflects the fact that Σ' is the final channel environment of a program.

Theorem 1 (Subject Reduction). If $\Gamma \vdash \Sigma \triangleright C \triangleleft \Sigma'$ and $C \rightarrow C'$, then $\Gamma \vdash \Sigma'' \triangleright C' \triangleleft \Sigma'$, for some Σ'' .

6 Type Safety

In our language of functional communicating threads different sorts of problems may occur at runtime, ranging from the traditional error of testing, in a

$$\frac{\Gamma \vdash \Sigma \triangleright t: \neg \triangleleft \Sigma'}{\Gamma \vdash \Sigma \triangleright \langle t \rangle \triangleleft \Sigma'} \qquad \frac{\Gamma \vdash \Sigma_1 \triangleright C_1 \triangleleft \Sigma'_1 \quad \Gamma \vdash \Sigma_2 \triangleright C_2 \triangleleft \Sigma'_2}{\Gamma \vdash \Sigma_1 \bullet \Sigma_2 \triangleright C_1 \mid C_2 \triangleleft \Sigma'_1 \bullet \Sigma'_2} \quad (T-THREAD, T-PAR)$$

$$\frac{\Gamma, n: [_] \vdash \Sigma \triangleright C \triangleleft \Sigma'}{\Gamma \vdash \Sigma \triangleright (\nu n) C \triangleleft \Sigma'} \qquad \frac{\Gamma \vdash \Sigma, c: \bot \triangleright C \triangleleft \Sigma' \quad c \text{ not in } \Sigma'}{\Gamma \vdash \Sigma \triangleright (\nu c) C \triangleleft \Sigma'} \quad (T-NewN, T-NewC)$$

Fig. 7. Typing rules for configurations

conditional expression, a value that is not true or false; through applying close to a value that is not a channel; to the most relevant to our work: having one thread trying to **send** on a given channel, and another trying to **select** on the same channel, or having three or more threads trying to synchronize on the same channel.

In order to define what we mean by a faulty configuration, we start by calling a *c*-thread any thread ready to perform an operation on channel *c*, that is a thread of the form (let x = receive c in t), and similarly for send, case, select, and close. A *c*-redex is the parallel composition of two threads ready to communicate on channel *c*, that is (let $x = \text{send } v \text{ on } c \text{ in } t_1$) | (let $y = \text{receive } c \text{ in } t_2$), and similarly for case/select, close/close. A configuration *C* is faulty when $C \equiv (\nu \tilde{n})(C_1 \mid C_2)$ and C_1 is

- 1. the thread (let x = e in t), where e is i) if v then _ else _ with $v \neq$ true, false, or is ii) v_{-} with $v \neq \lambda y.e'$ and $v \neq$ rec y.e'; or is
- 2. the thread (let x = accept/request v in t), where v is not a name;
- 3. the thread (let x = e in t), where e is i) receive/close v, or ii) send _on v, or iii) case v of _, or iv) select _on v, with v not a channel; or is
- 4. the parallel composition of two *c*-threads that do not form a *c*-redex; or is
- 5. the parallel composition of three or more *c*-threads.

Theorem 2 (Type Safety). Typable configurations are not faulty.

7 Related Work

Cyclone [8] is a C-like type-safe polymorphic imperative language. It features region-based memory management, and more recently threads and locks [7], via a sophisticated type system. The multithreaded version requires "a lock name for every pointer and lock type, and an effect for every function". Our locks are channels; but more than mutual exclusion, channels also allow a precise description of the protocol "between" acquiring and releasing the lock. In Cyclone a thread acquires a lock for a resource, uses the resource in whichever way it needs, and then releases the lock. Using our language a thread acquires the lock via a request operation, and then follows the protocol for the resource, before closing the channel obtained with request.

In the *Vault* system [2] annotations are added to C programs, in order to describe protocols that a compiler can statically enforce. Similarly to our approach, individual runtime objects are tracked by associating keys (channels, in our terminology) with resources, and function types describe the effect of the function on the keys. Although incorporating a form of selection (\oplus), the type system describes protocols in less detail than we can achieve with session types. "Adoption and Focus" [3], by the same authors, is a type system able to track changes in the state of objects; the system handles aliasing, and includes a form of polymorphism in functions. In contrast, our system checks the types of individual messages, as well as tracking the state of the channel. Our system is more

specialized, but the specialization allows more type checking in the situation that we handle.

Igarashi and Kobayashi have developed a generic framework in which a range of π -calculus type systems can be defined [12]. Although able to express sequencing of input and output types similarly to session types, it cannot express branching types.

A somewhat related line of research addresses resource access. Walker, Crary, and Morrisett [20] present a language to describe region-based memory management together with a provably safe type system. Igarashi and Kobayashi [11] present a general framework comprising a language with primitives for creating and accessing resources, and a type inference algorithm that checks whether programs access resources in a disciplined manner. Although types for resources in this latter work are similar in spirit to session types, we work in a much simpler setting.

Type and effect systems can be used to prove properties of protocols. Gordon and Jeffrey [6] use one such system to prove progress properties of communication protocols written in π -calculus. Rajamani *et al.*'s *Behave* [1,15] uses CCS to describe properties of π -calculus programs, verified via a combination of type and model checking. Since our system is purely type checking (not model checking) we expect verification to be more efficient and easier to implement.

8 Future Work

We outline some of the issues involved in extending our language to include a wider range of standard features.

Recursive Session Types. We have introduced recursive session types in a previous work [4]. We feel its incorporation in the present setting would not present major difficulties, although care must be taken in the definition of duality [19].

Principal Typings. For practical type inference, for separate compilation and modularity, one needs a notion of principal typings for the language. Particularly challenging is the share/not-share kind of polymorphism identified in section 2.

Type Inference. We are working on a constraint-based type inference algorithm for (the monomorphic fragment of) the language.

ML-style References and Assignment. This would introduce further issues of aliasing. We do not yet know whether our present infrastructure for type-checking in the presence of aliasing would be sufficient for this extension.

Acknowledgements. This work was partially supported by the EU IST proactive initiative FET-Global Computing (projects Mikado, IST–2001–32222, and Profundis, IST–2001–33100), Fundação para a Ciência e a Tecnologia (via CLC, CITI, and the project MIMO, POSI/CHS/39789/2001), and a Treaty of Windsor grant from the British Council in Portugal and the Portuguese Council of University Rectors.

References

- S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: model checking messagepassing programs. In *POPL*, pages 45–57. ACM Press, 2002.
- R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, pages 59–69. ACM Press, 2001.
- 3. J. S. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *PLDI*, pages 1–12, Berlin, Germany, June 2002.
- S. J. Gay, V. T. Vasconcelos, and A. Ravara. Session types for inter-process communication. TR 2003–133, Department of Computing Science, University of Glasgow, March 2003.
- S. J. Gay and M. J. Hole. Types and subtypes for client-server interactions. In S. D. Swierstra, editor, *ESOP*'99, LNCS 1576, pages 74–90. Springer-Verlag, 1999.
- 6. A. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300:379–409, 2003.
- 7. D. Grossman. Type-safe multithreading in cyclone. In ACM Workshop on Types in Language Design and Implementation. ACM Press, 2003.
- 8. D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Regionbased memory management in cyclone. In *PLDI* (pp. 282–293). ACM Press, 2002.
- 9. K. Honda. Types for dyadic interaction. In *CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer-Verlag, 1993.
- K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *ESOP'98*, volume 1381 of *LNCS*, pages 122–138. Springer-Verlag, 1998.
- A. Igarashi and N. Kobayashi. Resource usage analysis. In *POPL*, pages 331–342. ACM Press, 2002.
- 12. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1–3):121–163, 2003.
- R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. Information and Computation, 100(1):1–77, September 1992.
- 14. B. Pierce. Types and Programming Languages. The MIT Press, 2002.
- S. K. Rajamani and J. Rehof. A behavioral module system for the pi-calculus. In P. Cousot, editor, *Static Analysis: 8th International Symposium, SAS 2001*, volume 2126 of *LNCS*, pages 375–394. Springer-Verlag, 2001.
- J. Reppy. CML: a higher order concurrent language. In *PLDI*, pages 293–305. ACM Press, 1991.
- D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
- 18. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE '94*, LNCS 817. Springer-Verlag, 1994.
- 19. A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of objects and components using session types. In *FOCLASA 2002*, volume 68 of *Electronic Notes in Theoretical Computer Science*. Elsevier, August 2002.
- D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *TOPLAS*, 22(4):701–771, 2000.

A Higher Order Modal Fixed Point Logic

Mahesh Viswanathan¹ and Ramesh Viswanathan²

¹ University of Illinois at Urbana-Champaign vmahesh@uiuc.edu ² Bell Laboratories rv@research.bell-labs.com

Abstract. We present a higher order modal fixed point logic (HFL) that extends the modal μ -calculus to allow predicates on states (sets of states) to be specified using recursively defined higher order functions on predicates. The logic HFL includes negation as a first-class construct and uses a simple type system to identify the monotonic functions on which the application of fixed point operators is semantically meaningful. The model checking problem for HFL over finite transition systems remains decidable, but its expressiveness is rich. We construct a property of finite transition systems that is not expressible in the Fixed Point Logic with Chop [1] but which can be expressed in HFL. Over infinite transition systems, HFL can express bisimulation and simulation of push down automata, and any recursively enumerable property of a class of transition systems representing the natural numbers.

1 Introduction

An attractive methodology for compositional or heirarchical verification is the assumption-guarantee paradigm [2], in which a component of a system is specified in terms of assumptions it makes about its environment (other components), and properties it guarantees about its behavior, provided the assumptions hold. Using $\varphi \triangleright \psi$ to syntactically denote the property that under the assumptions φ , the property ψ is guaranteed, the semantics of the assume guarantee property needs to accomodate the following circular compositional rule: for a system $P = P_1 || P_2$, if P_1 satisfies the property $\varphi_2 \triangleright \varphi_1$ and P_2 satisfies the property $\varphi_1 \triangleright \varphi_2$ then P satisfies $\varphi_1 \land \varphi_2$. That such a semantics can be defined, for certain properties, was first observed by Misra and Chandy [3], and later formalized by Abadi and Lamport [4, 5]. Subsequently, it has been extended to other concurrency models and richer classes of properties [6, 7]. A unifying framework was provided in [8], in which the assume guarantee semantics was defined for properties expressible as fixed points; previously proposed rules then arise as instances of this framework.

To utilize the assume-guarantee paradigm in developing a formal system for compositional reasoning of concurrent programs, an obvious necessity is a logic or language in which the assume-guarantee semantics can be expressed. A natural candidate logic is the modal μ -calculus [9] which contains almost all other

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 512-528, 2004.

[©] Springer-Verlag Berlin Heidelberg 2004

temporal specification logics [10], and for which model-checking is decidable [11]. The distinguishing feature of the μ -calculus is the presence of fixed points that allow recursive definitions of predicates on states (subsets of states), thus corresponding exactly to the class of properties considered in [8]. But, as we show in Section 2, there are assume-guarantee properties that cannot be expressed in the μ -calculus, *i.e.*, assume-guarantee properties between two recursively defined subsets of states are not necessarily, in turn, expressible recursively. However, as also detailed in Section 2, there is a simple way to express all assume-guarantee properties by using a recursively defined function that takes as arguments subsets of states and returns a subset of states. Consequently, we were interested in a logic in which we can express recursively defined higher order functions.

A natural way to extend the modal μ -calculus to include higher order functions is to add the operations of λ -calculus, together with higher-order fixed points. The main technical difficulty with such an extension is a suitable accounting of nonmonotonic operators such as negation. In the modal μ -calculus, it can be assumed (without any loss of expressiveness) that negation is applied only to propositional constants and not to arbitrary formulas — in particular, negation cannot be applied to variables. This automatically ensures that formulas are monotonic in all their free variables thus assuring the semantic well-definedness of recursively binding them. Making such an assumption in the higher-order case would however be overly restrictive. The reason is that while the only intended use of variables in the modal μ -calculus is for recursive definitions, the extension with λ -abstraction also includes variables that are λ -bound, *i.e.*, used as formal parameters of functions which are semantically well-defined independent of being monotonic. Thus, restricting the use of negation, would force all definable functions to be monotonic, and it may lead to a loss in expressivity. In particular, the function that we use in expressing assume-guarantee properties is antimonotonic in one of its arguments, and furthermore requires the application of negation to a variable that stands for a formal function parameter. However, allowing the free use of negation means that formulas expressible in the logic can be of arbitrary monotonicity in their free variables and the logic needs to incorporate a systematic means of distinguishing semantically meaningful recursive definitions from invalid ones.

We present a logic that we call higher order fixed point logic (HFL). At the level of terms, HFL is a simple full-fledged union of propositional logic, modality operators, and λ -calculus with fixed point operators. It thus allows arbitrary use of negation and can accomodate recursive definitions of functions of arbitrary monotonicity. However, we formulate a type system that is an enrichment of the simply-typed λ -calculus which identifies the monotonicity of terms in their free variables and assures all well-typed terms to be semantically well-defined.

While the ability to define higher-order functions may be interesting, their impact on the class of definable predicates on states is not obvious *a priori*. The formulation of the logic HFL allows us to precisely explore this question. We consider the Fixed Point Logic with Chop (FLC), first proposed in [1], and further studied in [12, 13]. Formulas in FLC denote unary functions from sets of

states to sets of states and the logic includes fixed points allowing such predicate transformers to be expressed recursively. FLC is strictly more expressive than μ calculus and can express, for example, context-sensitive languages of finite linear processes. We exhibit a translation of FLC into HFL that preserves the semantics of formulas, thus showing that HFL is as expressive as FLC. We then construct a property of finite transition systems that is not expressible in FLC but which is shown to be definable in HFL. This shows that definable functions of more than one argument and higher-order contribute to increased expressivity. Similarly, it can be proved that HFL is strictly more expressive than every sublogic of HFL in which the number of arguments or order of functional arguments is restricted to any finite level, *i.e.*, the increased expressivity resulting from the order of the functions used continues through at all levels. Inspite of this richer expressiveness even over finite transition systems, model checking for the full logic HFL is decidable. Furthermore, properties of transition systems expressible in HFL are shown to be closed upto bisimulation. However, the satisfiability and validity problems are undecidable for FLC [1], and are therefore *a fortiori* undecidable for HFL as well.

The property shown to be provably not expressible in FLC is constructed by encoding FLC formulas as finite transition systems and diagonalizing over them. To our knowledge, the encoding is novel and provides the first inexpressibility result for FLC. On the other hand, this construction does not yield a particularly natural property although it is inherently dictated by the fact that FLC can express context sensitive languages and the only known proofs that certain languages are not context sensitive are ultimately based on diagonalization (c.f. [14]). Examples of more directly presentable properties expressible in HFL are: (a) Simulation and bisimulation of Push Down Automata (PDA) processes [15–17], and (b) Partial recursive functions and recursively enumerable properties over a class of infinite transition systems representing the natural number; due to space limitations, these constructions are not detailed in this paper. The expressibility of both these properties (as well as assume-guarantee properties) rely on recursively defined functions that take multiple arguments, and would therefore not be (directly) describable in FLC. Besides these examples, HFL may be well-suited for reasoning about higher-order concurrent programs that include procedural or object-oriented abstractions.

The rest of the paper is organized as follows. Section 2 details the motivating context of assume-guarantee properties. The syntax and semantics of HFL are defined in Section 3, and expressivity results for HFL are detailed in Section 4.

2 Motivation: Assume Guarantee Properties

Let *S* be the set of states of a transition system (formally defined in Section 3). In μ -calculus, the semantics of formulas are subsets of states, *i. e.*, for a μ -calculus formula φ , its semantics $[\![\varphi]\!] \in 2^S$ where 2^S denotes the powerset of *S*; a transition system satisfies a formula φ if its initial state belongs to $[\![\varphi]\!]$. Such subsets of states can be defined recursively as least or greatest fixed points of mono-

tonic functions $F: 2^S \to 2^S$; we use $\mu X.F(X)$ to denote the least fixed point and $\nu X.F(X)$ for the greatest fixed point. The Tarski-Knaster [18] construction approximates the fixed points through repeated iterations of F whose limit yields the desired fixed point. In the case of greatest fixed points, the k'th approximation, denoted by $[\nu X.F(X)]^k$, is defined inductively as $[\nu X.F(X)]^0 = S$, and $[\nu X.F(X)]^{k+1} = F([\nu X.F(X)]^k)$. Assume-guarantee specifications will be syntactically denoted by $\nu X.A(X) \triangleright \nu X.G(X)$, and their informal reading is that the guarantee specification $\nu X.G(X)$ is satisfied. The semantics of an assume-guarantee property $\nu X.A(X) \triangleright \nu X.G(X)$, given in Definition 1¹ below (from [8]), requires that if the environment ensures that the k'th approximation of A is satisfied, then the k + 1'st approximation of G must be satisfied.

Definition 1. For monotonic functions $A, G: 2^S \to 2^S$, $\nu X.A(X) \triangleright \nu X.G(X) \in 2^S$ is defined as $s \in \nu X.A(X) \triangleright \nu X.G(X)$ iff $\forall k \ge 0$. $s \in [\nu X.A(X)]^k \Rightarrow s \in [\nu X.G(X)]^{k+1}$.

The salient property of the semantics given by Definition 1 is that for any state *s* such that $s \in \nu(X).A(X) \triangleright \nu X.G(X)$ and $s \in \nu X.G(X) \triangleright \nu X.A(X)$ we have that $s \in \nu(X).A(X) \cap \nu(X).G(X)$ —this can be shown using induction [8].

We now show that the μ -calculus is not closed under assume-guarantee specifications. Consider the μ -calculus formulas $\varphi = \nu X \cdot X \wedge \langle a \rangle X$ and $\psi = \nu X \cdot X \wedge \langle a \rangle X$ (b)X which assert that there is a path where an a and b transition respectively is always enabled. A transition system then satisfies $\varphi \triangleright \psi$ iff it has the property that for every n, if there is a path of length n of a transitions from the initial state then there is a path of length n+1 of b transitions from the initial state. Viewing this as a property of computation trees (the unrolling of a transition system to yield a possibly infinite tree) we show that this is not a regular tree language. Since the set of computation trees associated with models satisfying a μ -calculus formula define a regular language, it follows that the property $\varphi \triangleright \psi$ cannot be expressed in μ -calculus. The intuition behind why a tree automaton cannot recognize this property is because the tree automaton will need to "remember" the length of the longest a-sequence and use this to check against the length of a b sequence in another part of the transition system. Since the *a* sequence can be of arbitrary length, the automaton does not have enough "memory" to do the necessary checks. Formalizing this argument yields the following proposition.

Proposition 1. There exist properties φ and ψ expressible in μ -calculus, such that $\varphi \triangleright \psi$ is not expressible in μ -calculus.

The properties φ, ψ that we have exhibited are in fact expressible in CTL as well. Hence Proposition 1 actually demonstrates that all the classical branching-time logics, *i.e.*, CTL, CTL*, and μ -calculus, cannot express all assume-guarantee specifications built from any formulas in the respective logic.

¹ This special instance of the more general definition is applicable to fixed points whose approximations converge within ordinal ω .

However, assume-guarantee properties can be expressed naturally using a recursively defined function on predicates of states. Writing $\neg X$ to denote the complement S - X for a set $X \in 2^S$, and $X \wedge Y$ to denote the intersection of sets X, Y, consider the function $AssGuar^{A,G}$ (where $A, G: 2^S \rightarrow 2^S$ monotonic) which takes two arguments $x, y \in 2^S$ and returns an element of 2^S , which is the greatest solution satisfying the following recursive definition:

$$AssGuar^{A,G}(x,y) \;\; = \;\; (\neg x \lor y) \land AssGuar^{A,G}(A(x),G(y))$$

The function $AssGuar^{A,G}(S_1, S_2)$ returns a set of states S_3 such that $s \in S_3$ iff $\forall k \ge 0.s \in A^k(S_1) \Rightarrow s \in G^k(S_2)$. By using S for S_1 and G(S) for S_2 , we can get the assume-guarantee property as $\nu X.A(X) \triangleright \nu X.G(X) = AssGuar^{A,G}(\mathsf{tt}, G(\mathsf{tt}))$, where we write tt to denote the set S.

The property $\nu X.A(X) \triangleright \nu X.G(X)$ can thus be written using a function on subsets of sets $(AssGuar^{A,G})$ that is not monotonic in one of its arguments (x) and which is recursively defined using a body that applies negation to one of its parameters (x). Because of the use of negation, it is not even clear that the recursive solution that we require for $AssGuar^{A,G}$ is semantically well-defined. This motivates the formulation of a logic in which such functions can be defined and their semantic validity can be established.

3 The Logic HFL

Similar to the μ -calculus, our logic will be interpreted over labelled transition systems. Let $\mathcal{P} = \{p, q, \ldots\}$ be a set of propositional constants, and $Act = \{a, b, \ldots\}$ be a set of action names. A labelled transition system is a structure $\mathcal{T} = (S, \{\stackrel{a}{\rightarrow} \mid a \in Act\}, L, s_0)$, where S is a set of states, $\stackrel{a}{\rightarrow}$ for each $a \in Act$ is a binary relation on states, $L: S \rightarrow 2^{\mathcal{P}}$ with L(s) for any $s \in S$ being the set of propositional constants that are true in state s, and $s_0 \in S$ is the state designated as the initial state. We use the infix notation $s \stackrel{a}{\rightarrow} t$ to denote that $(s, t) \in \stackrel{a}{\rightarrow}$. The transition system is finite if the set of states S is finite.

The types of the logic are given by the following grammar:

$$\begin{array}{ll} \text{(Variances)} & v ::= + \mid - \mid 0\\ \text{(Types)} & A ::= \mathsf{Prop} \mid A^v \to A \end{array}$$

We use letters A,B,... to range over types. Each type will be interpreted as a partially ordered set. The base type **Prop** intuitively represents the type of "properties" — its elements are subsets of states ordered by set inclusion. The elements of the type $A^v \rightarrow B$ are functions from A to B that respect the ordering on the type A in a manner given by the variance v — the type $A^+ \rightarrow B$ consists of functions that are monotonic with respect to the ordering on A, the type $A^- \rightarrow B$ consists of functions that are antimonotonic with respect to the ordering on A, and the type $A^0 \rightarrow B$ consists of arbitrary functions that are not required to be monotonic or antimonotonic. These intuitions are formalized in Definition 2 below, where for partial orders $\mathcal{A} = (A, \leq_A)$, $\mathcal{B} = (B, \leq_B)$, we use $\mathcal{A} \rightarrow \mathcal{B}$ to denote the partial order of monotone functions ordered pointwise, *i.e.*, the underlying set of $\mathcal{A} \rightarrow \mathcal{B}$ is $\{f: A \rightarrow B \mid \forall x, y \in A.x \leq_A y \Rightarrow f(x) \leq_B f(y)\}$ and the ordering relation given by $f \leq_{\mathcal{A} \rightarrow \mathcal{B}} g$ iff $\forall x \in A.f(x) \leq_B g(x)$.

Definition 2 (Semantics of Types).

- 1. For any binary relation $R \subseteq A \times A$ on a set A, define the binary relations $R^+, R^-, R^0 \subseteq A \times A$ as follows: $R^+ = R$, $R^- = \{(a, b) \mid (b, a) \in R\}$, and $R^0 = R^+ \cap R^-$. For any partial order $\mathcal{A} = (A, \leq_A)$, define the partial order \mathcal{A}^v as (A, \leq_A^v) , where $v \in \{+, -, 0\}$.
- 2. Let $\mathcal{T} = (S, \{\stackrel{a}{\rightarrow} \mid a \in Act\}, L, s_0)$ be a labelled transition system. The semantics $\mathcal{T}[\![A]\!]$ of any type A is a partial order defined by induction on the type A as:

$$\mathcal{T}\llbracket \mathsf{Prop} \rrbracket = (2^S, \subseteq)$$
$$\mathcal{T}\llbracket A^v \to B \rrbracket = (\mathcal{T}\llbracket A \rrbracket)^v \to \mathcal{T}\llbracket B \rrbracket$$

It is easily verified that for any partial order $\mathcal{A} = (A, \leq_A)$, the structure \mathcal{A}^{ν} is a partial order (*i.e.*, the relation \leq_A^{ν} is also reflexive, transitive, and antisymmetric) and since $\mathcal{A} \rightarrow \mathcal{B}$ is a partial order for any partial orders \mathcal{A}, \mathcal{B} , it follows that $\mathcal{T}\llbracket A \rrbracket$ is a well-defined partial order for any type A. Furthermore, the partial order $(2^S, \subseteq)$ is a complete lattice (with set unions and intersections giving joins and meets respectively) and for any partial order \mathcal{A} and complete lattice \mathcal{B} , the partial order $\mathcal{A} \rightarrow \mathcal{B}$ is a complete lattice (with joins and meets computed pointwise); it therefore follows that $\mathcal{T}\llbracket A \rrbracket$, for any type A, is a complete lattice. For a partial order \mathcal{A} that is a complete lattice, we use $\sqcup_{\mathcal{A}}$ and $\sqcap_{\mathcal{A}}$ to denote its join and meet operations, and $\perp_{\mathcal{A}}$ and $\top_{\mathcal{A}}$ to denote its least and greatest elements.

Let $\mathcal{V} = \{x, y, x_1, \ldots\}$ be a set of variable names. The terms of the logic are generated by the following grammar, where *p* ranges over the set of propositional constants \mathcal{P} , *x* ranges over the set of variable names \mathcal{V} , and *a* ranges over the set of action names *Act*.

$$\varphi::= \ \mathrm{ff} \mid p \mid x \mid \neg \varphi \mid \varphi \lor \varphi \mid \langle a \rangle \varphi \mid \lambda(x^v : A) \varphi \mid (\varphi \varphi) \mid \mu(x : A) \varphi$$

We use $\varphi, \psi, \varphi_1, \ldots$ to range over terms. In comparing with the propositional μ -calculus, the new term form $\lambda(x^{\nu}: A)\varphi$ corresponds to function definitions and $(\varphi \ \psi)$ denotes the value of the function φ on the argument ψ . Additionally, least fixed points $\mu(x: A)\varphi$ are now available at all types, with the type annotation A on the μ -bound variable indicating the type at which the fixed point is being taken. In the term $\lambda(x^{\nu}: A)\varphi$, the type annotation A corresponds to the expected type of its argument x and the variance annotation v corresponds to the expected variance of the function in its argument x. We identify terms upto renaming of bound variables (α -equivalence) with λ and μ being the variable-binding constructs, and use $\varphi[x \mapsto \psi]$ for the substitution of term ψ for variable x in the term φ (with suitable renaming of bound variables to avoid capturing free variables). Terms for propositional conjunction, [a] modalities, and greatest fixed points can

be derived and therefore not included in the syntax of primitive terms. Following standard λ -calculus conventions, we write $A_1^{v_1} \rightarrow \cdots \rightarrow A_n^{v_n} \rightarrow B$ to mean the type $A_1^{v_1} \rightarrow (\cdots \rightarrow (A_n^{v_n} \rightarrow B))$ and $\varphi \psi_1 \cdots \psi_n$ as shorthand for $(\cdots ((\varphi \psi_1) \psi_2) \cdots \psi_n)$.

We use a type system to identify a subset of the terms generated by the above grammar as being well-formed. Besides restricting the application of functional terms to arguments of the right type, the main purpose of the typing rules is to ensure that in a term $\mu(x; A)\varphi$, the term φ is monotonic in its μ -bound variable x to assure the existence of the least fixed point. The type system consists of proof rules for deriving judgements of the form $\Gamma \vdash \varphi: A$, where the context Γ is a sequence of the form $x_1^{v_1}: A_1, \ldots, x_n^{v_n}: A_n$ with variables x_1, \ldots, x_n all distinct and each variance annotation $v_i \in \{+, -, 0\}$. A derivable judgement $x_1^{v_1}: A_1, \ldots, x_n^{v_n}: A_n \vdash \varphi: A$ is read as consisting of two assertions: (1) if variables x_1, \ldots, x_n have types A_1, \ldots, A_n respectively then φ is a well-formed term of type A, and (2) the variance of the term φ in the variable x_i is given by the annotation v_i : if $v_i = +$ then φ is monotonic in x_i , if $v_i = -$ then φ is antimonotonic in x_i , and if $v_i = 0$ then nothing about the variance in x_i is asserted.

In defining the typing rules, we use the following notation. For a variance v, we define its negation v^- as: $+^- = -, -^- = +$, and $0^- = 0$. This definition is extended pointwise to contexts, so that for a context $\Gamma = x_1^{v_1} : A_1, \ldots, x_n^{v_n} : A_n$, the context Γ^- is defined to be $x_1^{v_1} : A_1, \ldots, x_n^{v_n} : A_n$. The type system is given in Table 1 and consists of axioms for ff, propositional constants, and variables, and inference rules for the remaining term constructs. As is to be expected, the proof rule (μ) requires the μ -bound variable to appear monotonically in the body. The most interesting typing rule is that for application which splits into three cases depending on the variance of the function being applied in its argument. It is most easily understood on the basis of the semantic requirement of derivable typing judgements given by the second part of Lemma 1 below. The typing rules are simple but account faithfully for some of the subtleties in the interaction of negation with variables of higher-order type. As a simple example, consider the term $\varphi \equiv (f(\neg x)) \lor (\neg z)$ which at first glance seems to

(false)	$\varGamma \vdash ff:Prop$	(prop)	$\varGamma \vdash p$: Prop
(var)	$\Gamma', x^v : A, \Gamma'' \vdash x : A \text{ if } v \in \{0, +\}$	(not)	$\frac{\Gamma^- \vdash \varphi: Prop}{\Gamma \vdash \neg \varphi: Prop}$
(or)	$\frac{\varGamma \vdash \varphi_1: Prop \varGamma \vdash \varphi_2: Prop}{\varGamma \vdash \varphi_1 \lor \varphi_2: Prop}$	$(\langle \rangle)$	$\frac{\Gamma \vdash \varphi: Prop}{\Gamma \vdash \langle a \rangle \varphi: Prop}$
(abs)	$\frac{\Gamma, x^{\upsilon} \colon A \vdash \varphi \colon B}{\Gamma \vdash \lambda(x^{\upsilon} \colon A)\varphi \colon A^{\upsilon} \to B}$	(μ)	$\frac{\varGamma, x^+ \colon A \vdash \varphi \colon A}{\varGamma \vdash \mu(x \colon A)\varphi \colon A}$
(app^+)	$\frac{\Gamma \vdash \varphi : A^+ \to B \Gamma \vdash \psi : A}{\Gamma \vdash (\varphi \; \psi) : B}$	(app^{-})	$\frac{\Gamma \vdash \varphi : A^- \to B \Gamma^- \vdash \psi : A}{\Gamma \vdash (\varphi \; \psi) : B}$
(app^0)	$\frac{\varGamma \vdash \varphi : A^0 \longrightarrow B \varGamma \vdash \psi : A \varGamma^- \vdash \psi : A}{\varGamma \vdash (\varphi \; \psi) : B}$		

 Table 1. Syntax of Higher Order Fixed Point Logic

have x appearing negatively. Following the typing rules (*var*) and (*not*), we can see that f appears positively and z appears negatively, but the variance in xdepends on the variance of the variable f in its argument type; it would in fact be a positive occurrence if f is antimonotonic. Indeed, using the typing rules, we can derive $f^+: \operatorname{Prop} \to \operatorname{Prop}, z^-: \operatorname{Prop}, x^+: \operatorname{Prop} \vdash \varphi: \operatorname{Prop}$, from which it follows that $\mu(f: \operatorname{Prop} \to \operatorname{Prop})\lambda(z^-: \operatorname{Prop})\mu(x: \operatorname{Prop})\varphi$ is a well-typed term of type $\operatorname{Prop} \to \operatorname{Prop}$ a recursive definition of an antimonotonic function.

The following proposition states a technically useful property of the type system, where we use \equiv to denote syntactic identity.

Proposition 2 (Unique Types). If $\Gamma \vdash \varphi$: A and $\Gamma \vdash \varphi$: A' are derivable, then $A \equiv A'$.

In particular, every closed term has a unique type. From Proposition 2 and the form of the typing rules (every construct has a unique rule for its introduction except for application but whose proof rule is uniquely determined by the type of the subterm), it follows that the derivation tree is unique as well (upto renaming of any bound variables). The proof of Proposition 2 is a straightforward induction on the length of the derivation, but it holds only because of the inclusion of the type and variance annotations for bound variables in the syntax of terms. For example, without type annotations, the term $\mu(x)x$ can be given any type A, and without variance annotations, the term $\lambda(x: \operatorname{Prop})x$ would have both the types $\operatorname{Prop}^+ \to \operatorname{Prop}$ and $\operatorname{Prop}^0 \to \operatorname{Prop}$.

We are now ready to define the semantics of terms. Let T be a transition system. An environment η is a possibly partial map on the variable set \mathcal{V} . For a context $\Gamma = x_1^{v_1}: A_1, \ldots, x_n^{v_n}: A_n$, we say that η is Γ -respecting, written $\eta \models \Gamma$ if $\eta(x_i) \in T[\![A_i]\!]$ for $i = 1, \ldots, n$. We write $\eta[x \mapsto a]$ for the environment that maps x to a and is the same as η on all other variables: if $\eta \models \Gamma$ and $a \in T[\![A]\!]$ for some type A then $\eta[x \mapsto a] \models \Gamma, x: A$, where x is a variable that does not appear in Γ . For any well-typed term $\Gamma \vdash \varphi: A$ and environment $\eta \models \Gamma$, Table 2 defines

Table 2. Semantics of Higher Order Fixed Point Logic

$$\begin{aligned} & \operatorname{For}\,\mathcal{T}=(S,\{\overset{a}{\rightarrow}\mid a\in Act\},L,s_{0})\\ & \mathcal{T}\llbracket\Gamma\vdash\operatorname{ff:}\operatorname{Prop}]\eta=\emptyset\\ & \mathcal{T}\llbracket\Gamma\vdash\operatorname{p:}\operatorname{Prop}]\eta=\{s\in S\mid p\in L(s)\}\\ & \mathcal{T}\llbracket\Gamma\vdash x:A]\eta=\eta(x)\\ & \mathcal{T}\llbracket\Gamma\vdash x:A]\eta=\eta(x)\\ & \mathcal{T}\llbracket\Gamma\vdash \neg\varphi\colon\operatorname{Prop}]\eta=S-\mathcal{T}\llbracket\Gamma\vdash\varphi\colon\operatorname{Prop}]\eta\\ & \mathcal{T}\llbracket\Gamma\vdash\varphi\lor\psi\colon\operatorname{Prop}]\eta=T[\Gamma\vdash\varphi\colon\operatorname{Prop}]\eta\cup\mathcal{T}\llbracket\Gamma\vdash\psi\colon\operatorname{Prop}]\eta\\ & \mathcal{T}\llbracket\Gamma\vdash\varphi\lor\psi\colon\operatorname{Prop}]\eta=\{s\in S\mid s\overset{a}{\rightarrow}t\text{ for some }t\in\mathcal{T}\llbracket\Gamma\vdash\varphi\colon\operatorname{Prop}]\eta\}\\ & \mathcal{T}\llbracket\Gamma\vdash\lambda(x^{v}:A)\varphi\colon A^{v}\to B]\eta=F\in\mathcal{T}\llbracketA^{v}\to B]\text{ s.t.}\\ & \forall d\in\mathcal{T}\llbracketA].F(d)=\mathcal{T}\llbracket\Gamma,x^{v}\colon A\vdash\varphi\colon B]\eta[x\mapsto d]\\ & \mathcal{T}\llbracket\Gamma\vdash\varphi\psi\colon B]\eta=\mathcal{T}\llbracket\Gamma\vdash\varphi\colon A^{v}\to B]\eta(\mathcal{T}\llbracket\Gamma\vdash\psi\colon A]\eta)\\ & \mathcal{T}\llbracket\Gamma\vdash\mu(x:A)\varphi\colon A]\eta=\sqcap_{\mathcal{T}\llbracketA}\{d\in A\mid\mathcal{T}\llbracket\Gamma,x^{+}\colon A\vdash\varphi\colon A]\eta[x\mapsto d]\leq_{\mathcal{T}\llbracket A}]d\}\end{aligned}$$

its semantics $\mathcal{T}\llbracket\Gamma \vdash \varphi: A
rbracket \eta$ to be an element of $\mathcal{T}\llbracketA
rbracket$. Referring to Table 2, in the case of the application term $(\varphi \ \psi)$, the type $A^v \rightarrow B$ is the unique type (as given by Proposition 2) of the term φ in the context Γ , the context Γ' is Γ if $v \in +, 0$, and is Γ^- if v = -.

For a context $\Gamma = x_1^{v_1}: A_1, \ldots, x_n^{v_n}: A_n$, we define the preorder relation \preceq_{Γ} on Γ -respecting environments as $\eta \preceq_{\Gamma} \chi$ iff $\eta(x_i) \leq_{T[A_i]}^{v_i} \chi(x_i)$ for $i = 1, \ldots, n$ (the relation \leq^v as given by Definition 2). We then show by induction on the typing derivation that the semantics of terms given in Table 2 is well-defined as an element of the appropriate type and is monotonic with respect to the preordering on the context.

Lemma 1 (Semantics of Terms). Let \mathcal{T} be a transition system. For any derivable $\Gamma \vdash \varphi$: A and environments $\eta, \chi \models \Gamma$, we have the following:

- 1. (Well-Definedness): $\mathcal{T}[\Gamma \vdash \varphi; A] \eta \in \mathcal{T}[A]$ and is uniquely defined.
- 2. (Variance): If $\eta \preceq_{\Gamma} \chi$ then $\mathcal{T}\llbracket \Gamma \vdash \varphi : A \rrbracket \eta \leq_{\mathcal{T}\llbracket A \rrbracket} \mathcal{T}\llbracket \Gamma \vdash \varphi : A \rrbracket \chi$.

Since for any closed term φ , there is a unique type A_{φ} such that $\emptyset \vdash \varphi : A_{\varphi}$ is derivable, we use $\mathcal{T}[\![\varphi]\!]$ to denote $\mathcal{T}[\![\emptyset \vdash \varphi : A_{\varphi}]\!]\emptyset$ where the environment \emptyset is undefined on all variables. Formulas are closed terms of type Prop, *i.e.*, terms φ such that $\emptyset \vdash \varphi$: Prop is derivable. As is standard, a transition system satisfies a formula, $\mathcal{T} \models \varphi$, iff the initial state $s_0 \in \mathcal{T}[\![\varphi]\!]$. A property of a class \mathcal{C} of transition systems is simply a subset $\mathcal{P} \subseteq \mathcal{C}$. A property \mathcal{P} of a class of transition systems \mathcal{C} is *expressible* if there is a characteristic formula $\varphi_{\mathcal{P}}$ such that for any $\mathcal{T} \in \mathcal{C}$, we have that $\mathcal{T} \models \varphi_{\mathcal{P}}$ iff $\mathcal{T} \in \mathcal{P}$. For a closed term φ , we write $\varphi : \mathcal{A}$ for the derivability of $\emptyset \vdash \varphi : \mathcal{A}$.

3.1 Invariance Under Bisimilarity

Satisfaction of any HFL formula by a transition system is invariant under bisimilarity of transition systems. This property cannot be established directly by induction because HFL formulas (closed terms of type Prop) can have subterms of higher-order type — we therefore need to suitably relate the semantics of higher-order terms in different transition systems. For transition systems $T = (S, \{\stackrel{a}{\rightarrow} \mid a \in Act\}, L, s_0)$ and $T' = (S', \{\stackrel{a}{\rightarrow}' \mid a \in Act\}, L', s'_0)$ and states $s \in S, s' \in S'$, we write $s \sim_{T,T'} s'$ to denote that s (with respect to T) is (labelrespecting) bisimilar to s' (with respect to T'), and $T \sim T'$ iff $s_0 \sim_{T,T'} s'_0$. For any type A, we define a binary relation $\mathcal{R}^A_{T,T'} \subseteq T[A] \times T'[A]$ by induction on the type A as follows (where we use the infix notation $a \mathcal{R}^A_{T,T'} a'$ to denote that $(a, a') \in \mathcal{R}^A_{T,T'}$):

$$P \mathcal{R}_{\mathcal{T},\mathcal{T}'}^{\mathsf{Prop}} P' \quad \text{iff} \quad \forall s \in S, s' \in S'. \ s \sim_{\mathcal{T},\mathcal{T}'} s' \Rightarrow (s \in P \text{ iff } s' \in P')$$
$$F \mathcal{R}_{\mathcal{T},\mathcal{T}'}^{A^{\oplus} \to B} F' \quad \text{iff} \quad \forall a \in \mathcal{T}[\![A]\!], a' \in \mathcal{T}'[\![A]\!]. \ a \mathcal{R}_{\mathcal{T},\mathcal{T}'}^{A} a' \Rightarrow F(a) \mathcal{R}_{\mathcal{T},\mathcal{T}'}^{B} F'(a')$$

For a context $\Gamma = x_1^{v_1}: A_1, \ldots, x_n^{v_n}: A_n$, define the relation $\mathcal{R}_{\mathcal{T},\mathcal{T}'}^{\Gamma}$ between \mathcal{T} and \mathcal{T}' -environments as $\eta \mathcal{R}_{\mathcal{T},\mathcal{T}'}^{\Gamma} \eta'$ iff $\eta(x_i) \mathcal{R}_{\mathcal{T},\mathcal{T}'}^{A_i} \eta'(x_i)$ for $i = 1, \ldots, n$. The following lemma establishes the connection between the semantics of higherorder terms in different models and is proved by induction on the structure of terms. As an immediate corollary, bisimilar transition systems satisfy the same set of HFL formulas.

Lemma 2. Let $\Gamma \vdash \varphi$: A be any derivable term. For any transition systems T, T' and respective environments η, η' with $\eta \mathcal{R}_{T,T'}^{\Gamma} \eta'$, we have that

$$\mathcal{T}\llbracket \Gamma \vdash \varphi : A \rrbracket \eta \ \mathcal{R}^{A}_{T,\mathcal{T}'} \ \mathcal{T}'\llbracket \Gamma \vdash \varphi : A \rrbracket \eta'$$

Corollary 1 (Bisimilarity Invariance). If $T \sim T'$ then for any formula φ , $T \models \varphi$ iff $T' \models \varphi$.

3.2 Model Checking

The model checking problem for HFL is decidable over finite state transition systems. This is an immediate consequence of the fact that for any finite transition system \mathcal{T} , the underlying set of $\mathcal{T}[\![B]\!]$ is finite for every type B. It therefore follows that $\mathcal{T}[\![\Gamma \vdash \varphi; A]\!]\eta$ can be computed inductively on the term φ (following the definition in Table 2) and using the standard iterative approximations to compute the semantics of fixed point terms. These iterative approximations for a fixed point term of type B converge after at most h_B iterations where h_B is the length of the longest strictly increasing chain in the partial order $\mathcal{T}[\![B]\!]$ (which is a finite number for any type B). This model-checking procedure is effective but not the most efficient; we leave exploration of other model-checking methods such as those based on tableaux to future work.

4 Expressiveness of HFL

Section 4.1 describes some basic definable operations in HFL that are used in developing the expressivity results, and Section 4.2 shows that HFL can express the assume-guarantee semantics of [8]. In Section 4.3, we show that the fixed point logic with chop (FLC)[1] can be translated into HFL so that any property expressible in FLC can also be expressed in HFL. In Section 4.4, we describe a representation of FLC formulas as transition systems over which we can diagonalize to construct properties inexpressible in FLC. In Section 4.5, we show that such a diagonalized property can be expressed in HFL, thereby establishing that HFL is strictly more expressive than FLC.

4.1 Definable Operations

Using standard dualities, we can define terms the three terms is of all states), $\varphi \wedge \psi$ (set intersection), and $[a]\varphi$; each of these terms is of type Prop and require φ, ψ to be of type Prop. Greatest fixed points, written $\nu(x; A)\varphi$, can be defined at arbitrary types A and require φ to be of type A with x appearing positively.

For any type A, we can define a closed term $bot_A: A$ denoting the least element at the type A. Call a transition system *finitely strongly connected* if it is strongly connected under transitions that have labels belonging to some finite set. Over a finitely strongly connected transition system, we can define functions on the type Prop by case-analysis. Let $\varphi_1, \ldots, \varphi_n$ be terms of type Prop, and $\psi_1, \ldots, \psi_n, \psi$ be terms of type $\operatorname{Prop}^0 \to A$ for some type A. We can define a term $\operatorname{case}^A(\varphi_1 \Rightarrow \psi_1, \ldots, \varphi_n \Rightarrow \psi_n, \operatorname{else} \Rightarrow \psi)$ of type $\operatorname{Prop}^0 \to A$ denoting a function that when applied to a singleton set $\{s\}$ (s is a state), returns $\llbracket \psi_i \rrbracket (\{s\})$ if $s \in \llbracket \varphi_i \rrbracket - \llbracket \varphi_{i-1} \rrbracket - \cdots - \llbracket \varphi_1 \rrbracket$ and returns $\llbracket \psi \rrbracket (\{s\})$ if $s \in S - \llbracket \varphi_n \rrbracket - \cdots - \llbracket \varphi_1 \rrbracket$. We use $\operatorname{case}^A(\varphi_1 \Rightarrow \psi_1, \ldots, \varphi_n \Rightarrow \psi_n)$ as syntactic sugar for the missing else clause returning $\operatorname{bot}_{\operatorname{Prop}^0 \to A}$. Note that by its very definition, the case-defined function cannot be monotonic or antimonotonic in its argument (of type Prop).

4.2 Assume Guarantee Properties

In this section, we show how assume-guarantee properties can be expressed in HFL. The encoding directly follows the informal recursive definition presented in Section 2; the main interest here is illustrating its well-typedness and its type. We define the closed term AssGuar as

$$\begin{array}{l} \lambda(f^-: \operatorname{Prop}^+ \to \operatorname{Prop})\lambda(g^+: \operatorname{Prop}^+ \to \operatorname{Prop}) \\ (\nu(z: \operatorname{Prop}^- \to \operatorname{Prop}^+ \to \operatorname{Prop})\lambda(x^-: \operatorname{Prop})\lambda(y^+: \operatorname{Prop})(\neg x \lor y) \land z \ (fx) \ (gy) \\) \ \operatorname{tt} \ (g \ \operatorname{tt}) \end{array}$$

which is typable as AssGuar: $(\operatorname{Prop}^+ \rightarrow \operatorname{Prop})^- \rightarrow (\operatorname{Prop}^+ \rightarrow \operatorname{Prop})^+ \rightarrow \operatorname{Prop}$. This typing judgement can be read as asserting that the assumption property (its first argument) and the guarantee property (its second argument) are required to be monotonic and that the assume-guarantee property itself varies antimonotonically in its assumption and monotonically in its guarantee. Constructing the type derivation for AssGuar is instructive in showing how these natural properties of AssGuar follow directly from the constraints imposed by the type system of Table 1.

The following proposition shows that the term AssGuar encodes assumeguarantee properties and establishes that HFL is closed under assume-guarantee specifications.

Proposition 3 (Expressibility of Assume-Guarantee). Consider any transition system T with state set S and any monotonic functions $A, G: 2^S \rightarrow 2^S$. Then we have that $T[AssGuar](A)(G) = \nu X.A(X) \triangleright \nu X.G(X)$.

4.3 Translating FLC into HFL

Let $\mathcal{T} = (S, \{\stackrel{a}{\rightarrow} \mid a \in Act\}, L, s_0), \mathcal{P}, Act, and \mathcal{V}$ be as described in Section 3. The following grammar describes the syntax of FLC formulas, where $p \in \mathcal{P}$, $a \in Act, x \in \mathcal{V}$.

$$\varphi \ ::= \ \text{ ff } \mid \text{tt} \mid p \mid \overline{p} \mid \varphi \lor \varphi \mid \varphi \land \varphi \mid \langle a \rangle \mid [a] \mid x \mid \mu x.\varphi \mid \nu x.\varphi \mid \text{term} \mid \varphi;\varphi$$

The formula \overline{p} is the negation of p; thus, negation in FLC is only applicable to propositional constants. Formulas are interpreted in FLC as predicate transformers, *i.e.*, functions $f: 2^S \rightarrow 2^S$ that are monotonic with respect to the subset ordering. The formula term denotes the identity function, and the chop operator; denotes function composition. An environment η for a formula φ is a map from variables to monotonic functions from 2^{s} to $2^{s'}$ that is defined on all the free variables of φ . For such an environment η , the FLC-semantics of a formula, written, $\mathcal{T}[\![\varphi]\!]^{\mathcal{C}}\eta$ yields a monotonic function from 2^{S} to 2^{S} . The reader is referred to [1] for the details of this definition, though it should also be clear from the translation into HFL that we next describe. A transition system satisfies a closed FLC formula, written $\mathcal{T} \models_{\mathcal{C}} \varphi$ iff $s_0 \in \mathcal{T}[\![\varphi]\!]^{\mathcal{C}} \emptyset(S)$, *i.e.*, the initial state is in the set obtained by applying the semantics to the full state set S. The superscript or subscript C refers to the semantics or satisfaction relation in the FLC logic.

Every FLC formula can be interpreted naturally as an HFL term of type **Prop**⁺ \rightarrow **Prop**. Table 3 details the straightforward inductive translation of any FLC formula φ into an HFL term $\hat{\varphi}$; it follows almost directly the semantics of FLC defined in [1]. The HFL term forms tt, $\varphi_1 \wedge \varphi_2$, $[a]\varphi$ and $\nu(x; A)\varphi$ used in the translation are the definable operations of Section 4.1, and the λ -bound variable z used in the translation of $\lor, \land, ;$ is one that does not appear free in the formula being translated. For an FLC formula φ , define the HFL context Γ_{φ} to be $x_1: \operatorname{Prop}^+ \to \operatorname{Prop}, \ldots, x_n: \operatorname{Prop}^+ \to \operatorname{Prop}$ for some enumeration x_1, \ldots, x_n of the free variables of φ . The following theorem shows that the translation is well-typed and preserves the semantics.

Theorem 1. For any FLC formula φ and transition system T, we have the following properties:

- 1. $\Gamma_{\varphi} \vdash \widehat{\varphi}$: **Prop**⁺ \rightarrow **Prop** is derivable. 2. For any FLC environment η for φ , we have that $\eta \models \Gamma_{\varphi}$ and

 $T\llbracket \Gamma_{\varphi} \vdash \widehat{\varphi} \colon \mathsf{Prop}^+ \to \mathsf{Prop} \rrbracket \eta = T\llbracket \varphi \rrbracket^{\mathcal{C}} \eta$

As a straightforward corollary, any property of transition systems expressed by an FLC formula φ can be expressed by the HFL formula $\widehat{\varphi}$ tt. From the

$\widehat{ff} = \lambda(z^+:Prop)ff$	$\widehat{\mathrm{tt}} = \lambda(z^+:Prop)tt$
$\widehat{p} = \lambda(z^+:Prop)p$	$\widehat{\overline{p}} = \lambda(z^+:Prop) \neg p$
$\widehat{x} = x$	
$\widehat{\langle a \rangle} = \lambda(z^+:Prop)\langle a \rangle z$	$\widehat{[a]} = \lambda(z^+: Prop)[a]z$
$\widehat{\varphi_1 \vee \varphi_2} = \lambda(z^+: Prop)(\widehat{\varphi_1} \ z) \vee (\widehat{\varphi_2} \ z)$	$\widehat{\varphi_1 \wedge \varphi_2} = \lambda(z^+ : Prop)(\widehat{\varphi_1} \ z) \wedge (\widehat{\varphi_2} \ z)$
$\widehat{term} = \lambda(z^+:Prop)z$	$\widehat{\varphi_1;\varphi_2} = \lambda(z^+:Prop)\widehat{\varphi_1}(\widehat{\varphi_2}\ z)$
$\widehat{\mu x.\varphi} = \mu(x:Prop^+{\rightarrow}Prop)\widehat{\varphi}$	$\widehat{\nu x.\varphi} = \nu(x:Prop^+{\rightarrow}Prop)\widehat{\varphi}$

Table 3. Translation of FLC into HFL

results established in [1], it then also follows that satisfiability and validity of HFL formulas is undecidable.

Corollary 2. For any transition system T and closed FLC formula φ , we have that $T \models \widehat{\varphi}$ tt iff $T \models_{c} \varphi$.

4.4 Properties Inexpressible in FLC

Define the set $SF(\varphi)$ of subformulas of any FLC formula φ in the standard way with $SF(\sigma x.\varphi) = \{\sigma x.\varphi\} \cup SF(\varphi)$, where σ is μ or ν . Call an FLC formula φ well-named if each bound variable in the formula φ is distinct. In this case, there is a well-defined function $FP_{\varphi}: (\mathcal{V} \cap SF(\varphi)) \rightarrow SF(\varphi)$ that maps each variable $x \in SF(\varphi)$ to a unique formula of the form $\sigma x.\psi \in SF(\varphi)$ where σ is μ or ν . We identify four action names from the set Act which we call lc, rc, ev, and dm and let $A = \{lc, rc, ev, dm\}$. These four names can be read as "left child", "right child", "evaluation", and "dummy". We also identify propositional constants from \mathcal{P} that we will refer to by $p_l, p_v, p_{\wedge}, p_{\mu}, p_{\nu}, p_{\text{term}}, p_i$, and $p_{\langle a \rangle}, p_{[a]}$ for each $a \in A$.

We now give our representation of FLC formulas as labelled transition systems.

Definition 3. For any well-named FLC formula φ whose action names all belong to the set $A = \{lc, rc, ev, dm\}$, the transition system \mathcal{T}^{φ} is defined to be $(SF(\varphi), \{\stackrel{a}{\rightarrow}\}, L, \varphi)$ where:

- The labelling function is defined according to the form of the formula: $L(\psi) = \{p_l\}$ if ψ is one of \mathfrak{t} , \mathfrak{f} , p,\overline{p} for some $p \in \mathcal{P}$; $L(x) = \{p_v\}$; $L(\psi) = \{p_{\psi}\}$ if ψ is of the form term, $\langle a \rangle$ or [a] for $a \in A$; $L(\psi_1 O \psi_2) = \{p_O\}$ where O is one of $\lor, \land, ;$, and $L(\sigma x.\psi) = \{p_\sigma\}$ where σ is μ or v.
- For any action name $a \notin A$, $\stackrel{a}{\rightarrow} = \emptyset$. The set $\stackrel{lc}{\rightarrow}$ includes pairs (ψ_1, ψ_2) where ψ_2 is of the form $\psi_1 O \psi'$, for some ψ' , or $\sigma x.\psi_1$, where O is one of $\lor, \land, ;$, and σ is one of μ , v. The set $\stackrel{rc}{\rightarrow}$ includes pairs (ψ_1, ψ_2) where ψ_2 is of the form $\psi' O \psi_1$, for some ψ' and O one of $\lor, \land, ;$. The set $\stackrel{ev}{\rightarrow}$ includes pairs (ψ_1, ψ_2) which satisfy one of the four conditions: (1) ψ_2 is the formula tt, (2) ψ_2 is p for some $p \in \mathcal{P}$, and $p \in L(\psi_1)$, (3) ψ_2 is \overline{p} for some $p \in \mathcal{P}$, and $p \notin L(\psi_1)$, (4) ψ_2 is a variable x and ψ_1 is $FP_{\varphi}(x)$. Finally, the set $\stackrel{dm}{\rightarrow}$ includes pairs (φ, ψ) (where φ is the formula being represented) and ψ is one of tt, ff, $p, \overline{p}, x, \langle a \rangle, [a]$ for $a \in A$.

The transition system \mathcal{T}^{φ} is finite and strongly connected by the transitions from A.

Definition 3 can be intuitively understood as follows. The transition system for a formula φ is essentially its parse tree (with sharing of the trees for common subformulas) with edges directed from child to parent. These parse tree edges are labelled with the lc, rc transitions, and the propositional labeling indicates the outermost construct of the corresponding subformula (with p_l standing for constant literals and p_v for variables). Additionally, we have transitions labeled ev to the constant literals tt, ff, p, \overline{p} from all the states in which the literals hold (ff does not hold anywhere), and to variables x from their defining fixed point formula. Note that because the formula is well-named, there is exactly one transition labeled ev to every node x. Finally, the dummy transition edges dm are added from the root to every leaf node — the only purpose of these edges is to make the transition system strongly connected (thus allowing us to use the **case**-construct over these transition systems). It also allows us to identify the initial state (as the only one that has a dm transition enabled).

By diagonalizing over this representation, we obtain properties of finite transition systems that cannot be expressed in FLC.

Theorem 2. Let C be any property of finite transition systems such that for any closed well-named formula φ with actions from A, we have that $\mathcal{T}^{\varphi} \in C$ iff $\mathcal{T}^{\varphi} \not\models_{C} \varphi$. Then C is not expressible in FLC.

Note that the inexpressible property described by Theorem 2 is unconstrained on transition systems that are not a T^{φ} .

4.5 HFL Is More Expressive Than FLC

We now show how to construct a formula in HFL that expresses a property of the form prescribed by Theorem 2. Table 4 defines HFL terms whose types are as follows:

Table 4.	Encoding	FLC	Diagonalization	ı in	HFL
----------	----------	-----	-----------------	------	-----

$$\begin{array}{l} \operatorname{decode} \stackrel{\Delta}{=} \mu(d:(\operatorname{Prop}^{0} \rightarrow (\operatorname{Prop}^{+} \rightarrow \operatorname{Prop}))^{+} \rightarrow (\operatorname{Prop}^{0} \rightarrow (\operatorname{Prop}^{+} \rightarrow \operatorname{Prop}))) \\ \lambda(e^{+}:(\operatorname{Prop}^{0} \rightarrow (\operatorname{Prop}^{+} \rightarrow \operatorname{Prop}))) \\ \operatorname{case}(p_{l} \Rightarrow \lambda(x^{0}:\operatorname{Prop})\lambda(z^{+}:\operatorname{Prop}) \langle ev \rangle x, \\ p_{v} \Rightarrow \lambda(x^{0}:\operatorname{Prop})\lambda(z^{+}:\operatorname{Prop}) \langle a \rangle z \stackrel{a \in \{\operatorname{lc}, \operatorname{rc}, \operatorname{ev}, \operatorname{dm}\}}{p_{\langle a \rangle}}, \\ p_{\langle a \rangle} \Rightarrow \lambda(x^{0}:\operatorname{Prop})\lambda(z^{+}:\operatorname{Prop}) [a] z \stackrel{a \in \{\operatorname{lc}, \operatorname{rc}, \operatorname{ev}, \operatorname{dm}\}}{p_{\langle \gamma}}, \\ p_{\vee} \Rightarrow \lambda(x^{0}:\operatorname{Prop})\lambda(z^{+}:\operatorname{Prop}) (d e (\langle \operatorname{lc} \rangle x) z) \lor (d e (\langle \operatorname{rc} \rangle x) z), \\ p_{\wedge} \Rightarrow \lambda(x^{0}:\operatorname{Prop})\lambda(z^{+}:\operatorname{Prop}) (d e (\langle \operatorname{lc} \rangle x) z) \land (d e (\langle \operatorname{rc} \rangle x) z), \\ p_{\operatorname{term}} \Rightarrow \lambda(x^{0}:\operatorname{Prop})\lambda(z^{+}:\operatorname{Prop}) (d e (\langle \operatorname{lc} \rangle x)) ((d e (\langle \operatorname{rc} \rangle x) z), \\ p_{\mu} \Rightarrow \lambda(x^{0}:\operatorname{Prop})\lambda(z^{+}:\operatorname{Prop}) (d e (\langle \operatorname{lc} \rangle x)) ((d e (\langle \operatorname{rc} \rangle x)) z), \\ p_{\mu} \Rightarrow \lambda(x^{0}:\operatorname{Prop}) \mu(f:\operatorname{Prop}^{+} \rightarrow \operatorname{Prop}) \\ d (\operatorname{case}(x \Rightarrow \lambda(z^{0}:\operatorname{Prop})f, \operatorname{else} \Rightarrow e)) (\langle \operatorname{lc} \rangle x) \\) \\ \operatorname{init} \stackrel{\Delta}{=} \langle \operatorname{dm} \rangle \operatorname{tt} \\ \operatorname{flc-sem} \stackrel{\Delta}{=} \operatorname{decode} \operatorname{bot}_{\operatorname{Prop}^{0} \rightarrow (\operatorname{Prop}^{+} \rightarrow \operatorname{Prop}) \operatorname{init} \\ \operatorname{flc-sem} \stackrel{\Delta}{=} - (\operatorname{flc-sem} \operatorname{tt}) \end{array}$$
decode : $(Prop^0 \rightarrow (Prop^+ \rightarrow Prop))^+ \rightarrow (Prop^0 \rightarrow (Prop^+ \rightarrow Prop))$ init : Prop flc-sem : $Prop^+ \rightarrow Prop$ flc-diag : Prop

with the HFL formula flc-diag expressing a property of the form prescribed by Theorem 2. The properties of the terms **decode** and init defined in Table 4 are given by the following theorem:

Theorem 3. Let φ be a closed well-named FLC formula over the action set A.

 Consider any subformula ψ ∈ SF(φ) and FLC environment η: V→(2^S→2^S). For any function F_η ∈ T^φ [Prop⁰→(Prop⁺→Prop)] such that F_η({FP_φ(x)}) = η(x) for every x free in ψ, T^φ [decode] (F_η)({ψ}) = T^φ [ψ]^c η
T^φ [init] = {φ}

The heart of the construction is decode that shows how to decode (in HFL) the transition system representing an FLC formula. Its definition given in Table 4 is easiest understood on the basis of its property given in Theorem 3, with the λ -bound variable e read as standing for the function (F_n) representing an environment η , and the λ -bound variable x in each of the cases read as standing for the singleton set $\{\psi\}$. On an argument $\{\psi\}$, the formula ψ is decoded in cases according to its outermost form which in turn is inferred based on which of the propositional constants p_l, p_v, \ldots holds in x (standing for $\{\psi\}$). For all constructs other than variables and fixed points, their corresponding cases can be understood by close analogy with the HFL-translation of these constructs given in Table 3 together with the understanding that (lc)x and (rc)x yield singleton sets including the corresponding subformulas of ψ , and that for constant literals term $\langle ev \rangle x$ yields the set of states in which the literal ψ holds. If ψ is a variable, we evaluate the environment on the set $\{FP_{\varphi}(\psi)\}$ (as given by the property of F_n) which is yielded by the term $\langle ev \rangle x$. If ψ is a fixed point formula, we correspondingly bind (using μ or ν) a new variable f and decode the subformula of ψ (given by $\langle lc \rangle x$) but in an environment that is obtained by modifying the current environment e to map $\{\psi\}$ (given by x) to f (the case-term used for the environment argument to d in the fixed point cases yields this updated environment). This ensures that when decoding the subformulas of ψ any use of a variable corresponding to this recursive definition will be decoded as f. The decoding of the fixed-point cases explains the presence of the environment argument in defining decode. Finally, it is worth noting that: (1) decode is a recursive definition of a higher-order function, and (2) because decode is defined by case-analysis, it is not monotonic in the argument x (standing for the formula being decoded). These features of HFL are therefore crucial to its definition.

As an easy corollary of Theorem 3, we get the relevant properties of the terms flc-sem and flc-diag.

Corollary 3. For any closed well-named FLC formula φ over the action set A, we have that

- 1. T^{φ} [[flc-sem]] = T^{φ} [[φ]]^{\mathcal{C}} \emptyset . 2. $T^{\varphi} \models$ flc-diag $_{if}T^{\varphi} \not\models_{\mathcal{C}} \varphi$.

Combined with Theorem 2 this gives us that the HFL formula flc-diag is a characteristic formula for a property of finite transition systems that is inexpressible in FLC, and thus HFL is strictly more expressive than FLC even over finite transition systems.

References

- 1. Müller-Olm, M.: A Modal Fixpoint Logic with Chop. In: Proceedings of the Symposium on the Theoretical Aspects of Computer Science Volume 1563 of Lecture Notes in Computer Science., Springer (1999) 510-520
- 2. Pnueli. A .: In transition from global to modular temporal reasoning about programs. In: Logics and Models of Concurrent Systems. NATO ASI Series. Springer-Verlag (1984) 123-144
- 3. Misra, J., Chandy, K.M.: Proofs of network processes. IEEE Transactions on Software Engineering SE-7 (1981) 417-426
- 4. Abadi, M., Lamport, L.: Composing specifications. ACM Transactions on Programming Languages and Systems 15 (1993) 73-132
- 5. Abadi, M., Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages and Systems 17 (1995) 507-534
- 6. McMillan, K.: Circular compositional reasoning about liveness. In Pierre, L., Kropf, T., eds.: CHARME 99: Correct Hardware Design and Verification. Volume 1703 of Lecture Notes in Computer Science., Springer-Verlag (1999) 342-345
- 7. Henzinger, T.A., Qadeer, S., Rajamani, S.K., Tasiran, S.: An assume-guarantee rule for checking simulation. In: Gopalakrishnan, G., and Windley, P., eds: FMCAD 98: Formal Methods in Computer-aided Design, Volume 1522 of Lecture Notes in Computer Science., Springer-Verlag (1998) 421-432
- 8. Viswanathan, M., Viswanathan, R.: Foundations of Circular Compositional Reasoning. In: Proceedings of the International Colloquim on Automata, Languages and Programming. Lecture Notes in Computer Science, Springer (2001)
- 9. Kozen, D.: Results on the propositional μ -calculus. Theoretical Computer Science, 27 (1983) 333-354
- 10. Stirling, C.: Modal and Temporal Logics. In Handbook of Logic in Computer Science. Volume 2. Claredon Press, Oxford, UK (1992) 477-563
- 11. Emerson, E.A., Jutla, C.S., Sistla, A.P.: On model checking for the *µ*-calculus and its fragments. Theoretical Computer Science 258 (2001) 491-522
- 12. Lange, M., Stirling, C.: Model Checking Fixed Point Logic with Chop. In: Proceedings of the Foundations of Software Science and Computation Structures. Volume 2303 of Lecture Notes in Computer Science., Springer (2002) 250-263
- 13. Lange, M.: Local model checking games for fixed point logic with chop. In: Proceedings of the Conference on Concurrency Theory, CONCUR'02. Volume 2421 of Lecture Notes in Computer Science., Springer (2002) 240-254
- 14. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
- 15. Burkart, O. Caucal, D., Moller, F., Steffen, B.: Verification on Infinite Structures. In: Handbook of Process Algebra. Elsevier Science Publishers (2001) 545-623

- Hirshfeld, Y., Jerrum, M., Moller, F.: A polynomial algorithm for deciding bisimilarity of normed context-free processes. Theoretical Computer Science 15 (1996) 143–159
- Sénizergues, G.: Decidability of bisimulation equivalence for equations graphs of finite out-degree. In: Proceedings of the IEEE Sysmposium on the Foundations of Computer Science. (1998) 120–129
- Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics 5 (1955) 285–309

Author Index

Abdulla, Parosh Aziz 35 Amadio, Roberto M. 68 Andrews, Tony 1 Baldan, Paolo 83 Baudru, Nicolas 99 Berger, Martin 115 Bojańczyk, Mikołaj 131 Bollig, Benedikt 146 Borgström, Johannes 161 Bozga, Liana 177 Brázdil, Tomáš 193 Briais, Sébastien 161 Brookes, Stephen 16 Bruns, Glenn 209 Bugliesi, Michele 225 Caires, Luís 240 Cîrstea, Corina 258 Clarke, Edmund 276 Colazzo, Dario 225 Corradini, Andrea 83 Crafa, Silvia 225 Dal Zilio, Silvano 68 Danos, Vincent 292 Ene, Cristian 177 Gay, Simon 497 Groote, Jan Friso 308 Hirschkoff, Daniel 325 Jagadeesan, Radha 209 Jeffrey, Alan 209 Jonsson, Bengt 35 König, Barbara 83 Krčál, Pavel 340 Křetínský, Mojmír 355 Krivine, Jean 292 Kučera, Antonín 193, 371 Lakhnech, Yassine 177 Laroussinie, F. 387 Leroux, Jérôme 402 Leucker. Martin 146 Lozes, Étienne 240

Ma, Qin 417 Maranget, Luc 417 Markey, Nicolas 387, 432 Melliès, Paul-André 448 Mokrushin, Leonid 340 Morin. Rémi 99 Nestmann. Uwe 161 Nilsson, Marcus 35 O'Hearn, Peter W. 49 Pattinson, Dirk 258 Oadeer, Shaz 1 Rajamani, Sriram K. 1 Raskin, Jean-François 432 Ravara, António 497 Řehák, Vojtěch 355 Rehof, Jakob 1 Riely, James 209 Saksena, Mayank 35 Schnoebelen, Philippe 371, 387 Stražovský, Oldřich 193 Strejček, Jan 355 Sutre, Grégoire 402 Tabuada, Paulo 466 Talupur, Muralidhar 276 Thiagarajan, P.S. 340 Touili, Tayssir 276 Varacca, Daniele 481 Vasconcelos, Vasco 497 Veith, Helmut 276 Viswanathan. Mahesh 512 Viswanathan, Ramesh 512 Völzer, Hagen 481 Walukiewicz, Igor 131 Willemse, Tim 308 Winskel, Glynn 481 Xie. Yichen 1 Yi, Wang 340

This page intentionally left blank

This page intentionally left blank

This page intentionally left blank