Farhad Arbab
Marjan Sirjani (Eds.)

LNCS 5961

Fundamentals of Software Engineering

Third IPM International Conference, FSEN 2009
Kish Island, Iran, April 2009
Revised Selected Papers

Springer

Lecture Notes in Computer Science 5961

Farhad Arbab   Marjan Sirjani (Eds.)

Fundamentals of Software Engineering

Third IPM International Conference, FSEN 2009
Kish Island, Iran, April 15-17, 2009
Revised Selected Papers

Springer

Volume Editors

Farhad Arbab
Center for Mathematics and Computer Science (CWI)
Science Park 123, 1098 XG Amsterdam, The Netherlands
and Leiden University, The Netherlands
E-mail: farhad@cwi.nl

Marjan Sirjani
Reykjavik University, School of Computer Science
Kringlan 1, 103 Reykjavik, Iceland
and University of Tehran, Iran
E-mail: marjan@ru.is

Library of Congress Control Number: 2009942995

CR Subject Classification (1998): D.2, D.2.4, F.4.1, D.2.2

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN	0302-9743
ISBN-10	3-642-11622-1 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-11622-3 Springer Berlin Heidelberg New York

springer.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12840984 06/3180 5 4 3 2 1 0

Preface

The present volume contains the proceedings of the Third IPM International Conference on Fundamentals of Software Engineering (FSEN), Kish, Iran, April 15–17, 2009. FSEN 2009 was organized by the School of Computer Science at the Institute for Studies in Fundamental Sciences (IPM) in Iran, in cooperation with the ACM SIGSOFT and IFIP WG 2.2.

This conference brought together around 100 researchers and practitioners working on different aspects of formal methods in software engineering from 15 different countries. The topics of interest in FSEN span over all aspects of formal methods, especially those related to advancing the application of formal methods in software industry and promoting their integration with practical engineering techniques. The Program Committee of FSEN 2009 consisted of top researchers from 24 different academic institutes in 11 countries. We received a total of 88 submissions from 25 countries out of which the Program Committee selected 22 as regular papers, 5 as short papers, and 7 as poster presentations in the conference program. Each submission was reviewed by at least three independent referees, for its quality, originality, contribution, clarity of presentation, and its relevance to the conference topics. This volume contains the revised versions of the regular and short papers presented at FSEN 2009.

Three distinguished keynote speakers delivered their lectures at FSEN 2009 on models of computation: automata and processes (Jos Baeten), verification, performance analysis and controller synthesis for real-time systems (Kim Larsen), and theory and tool for component-based model-driven development in rCOS (Zhiming Liu). Our invited speakers also contributed to this volume by submitting their keynote papers, which were accepted after they were reviewed by independent referees.

We thank the Institute for Studies in Fundamental Sciences (IPM), Tehran, Iran for their financial support and local organization of FSEN 2009. We thank the members of the Program Committee and the external reviewers for their time, effort, and contributions to making FSEN a quality conference. We are grateful for the help of Behnaz Changizi in preparing the pre-proceedings of FSEN 2009 and this volume. Last but not least, our thanks go to our authors and conference participants, without whose submissions and participation FSEN would not have been possible.

November 2009

Farhad Arbab
Marjan Sirjani

Conference Organization

General Chair

Hamid Sarbazi-azad	IPM, Iran; Sharif University of Technology, Iran

Steering Committee

Farhad Arbab	CWI, The Netherlands; Leiden University, The Netherlands
Christel Baier	University of Dresden, Germany
Frank de Boer	CWI, The Netherlands; Leiden University, The Netherlands
Ali Movaghar	IPM, Iran; Sharif University of Technology, Iran
Jan Rutten	CWI, The Netherlands; Vrije University Amsterdam, The Netherlands
Hamid Sarbazi-azad	IPM, Iran; Sharif University of Technology, Iran

Marjan Sirjani	Reykjavik University, Reykjavik, Iceland;
	University of Tehran, Iran;
	IPM, Iran

Program Chairs

Farhad Arbab	CWI, The Netherlands; Leiden University, The Netherlands

Marjan Sirjani	Reykjavik University, Reykjavik, Iceland;
	University of Tehran, Iran;
	IPM, Iran

Program Committee

Luca Aceto	Reykjavik University, Reykjavik, Iceland
Gul Agha	University of Illinois at Urbana - Champaign, USA
Christel Baier	University of Dresden, Germany
Frank de Boer	CWI, The Netherlands; Leiden University, The Netherlands

Marcello Bonsangue	Leiden University, The Netherlands
Mario Bravetti	University of Bologna, Italy
Michael Butler	University of Southampton, UK
James C. Browne	University of Texas at Austin, USA
Dave Clarke	CWI, The Netherlands; K.U. Leuven, Belgium
Nancy Day	University of Waterloo, Canada
Wan Fokkink	Vrije Universiteit Amsterdam, The Netherlands
Masahiro Fujita	University of Tokyo, Japan
Maurizio Gabbrielli	University of Bologna, Italy
Jan Friso Groote	Technical University of Eindhoven, The Netherlands
Einar Broch Johnsen	University of Oslo, Norway
Joost Kok	Leiden University, The Netherlands
Zhiming Liu	United Nations University, Macao, China
Seyyed Hassan Mirian	Sharif University of Technology, Iran
Ugo Montanari	University of Pisa, Italy
Peter Mosses	Swansea University, UK
Mohammad Reza Mousavi	Technical University of Eindhoven, The Netherlands
Ali Movaghar	IPM, Iran; Sharif University of Technology, Iran
Andrea Omicini	University of Bologna, Italy
Jan Rutten	CWI, The Netherlands; Vrije University Amsterdam, The Netherlands
Davide Sangiorgi	University of Bologna, Italy
Sandeep Shukla	Virginia Tech, USA
Carolyn Talcott	SRI International, USA
Zijiang Yang	Western Michigan University, USA

Local Organization

Hamidreza Shahrabi	IPM, Iran

External Reviewers

Ahuja, Sumit
Amato, Gianluca
Andrei, Oana
Astefanoaei, Lacramioara
Atif, Muhammad
Bacciu, Davide
Bartocci, Ezio
Behjati, Razieh
Birgisson, Arnar
Bistray, Denes
Blechmann, Tobias
Blom, Stefan
Bokor, Peter
Browne, James

Bundgaard, Mikkel
Cacciagrano, Diletta Romana
Callanan, Sean
Carbone, Marco
Chen, Qichang
Chen, Zhenbang
Chessa, Stefano
Chiniforooshan Esfahani, Hesam
Chockler, Hana
Chothia, Tom
Costa, David
Crouzen, Pepijn
Cuijpers, Pieter
De Nicola, Rocco
Di Berardini, Maria Rita
Di Giusto, Cinzia
Dovland, Johan
Edmunds, Andrew
Eker, Steven
Fantechi, Alessandro
Ferreira, Carla
Ferretti, Stefano
Fuentes, Thaizel
Gerrits, Dirk
Ghassemi, Fatemeh
Grabe, Immo
Griesmayer, Andreas
Groesser, Marcus
Gruener, Andreas
Hugues, Jerome
Haghighi, Hassan
Hallerstede, Stefan
Hansen, Henri
Hasegawa, Masahito
Hoenicke, Jochen
Huang, Xiaowan
Izadi, Mohammad
Izadi, Mohammad Javad
Jaeger, Manfred
Jagannath, Vilas
Jaghoori, Mohammad Mahdi
Jahangard, Amir
Jose, Bijoy
Kane, Kevin
Karmani, Rajesh
Kemper, Stephanie
Khosravi, Ramtin
Kim, Minyoung
Kleijn, Jetty
Klein, Joachim
Klint, Paul
Klueppelholz, Sascha
Koehler, Christian
Korthikanti, Vijay Anand Reddy
Kyas, Marcel
Li, Dan
Lin, Cui
Lluch Lafuente, Alberto
Luettgen, Gerald
Maamria, Issam
Magnani, Matteo
Mathaikutty, Deepak
Mathijssen, Aad
Montaghami, Vajihollah
Montangero, Carlo
Mooij, Arjan
Morisset, Charles
Nangia, Saurabh
Nanz, Sebastian
Orzan, Simona
Osaiweran, Ammar
Patel, Hiren
Polini, Andrea
Razavi, Niloofar
Ren, Shangping
Reniers, Michel
Rezazadeh, Abdolbaghi
Riganelli, Oliviero
Ripon, Shamim
Roggenbach, Markus
Rossi, Davide
Said, Mar Yah
Sabouri, Hamideh
Schlatte, Rudolf
Shali, Amin
Silva, Alexandra
Snidaro, Lauro
Snook, Colin
Stappers, Frank
Steffen, Martin

Stehr, Mark-Oliver
Stolz, Volker
Suhaib, Syed
Sun, Meng
Tartamella, Chris
Tesei, Luca
Tuosto, Emilio
Turini, Franco
Voorhoeve, Marc
Wang, Chao
Wang, Shuling
Wang, Xu
Willemse, Tim
Wu, Zeng
van der Wulp, Jeroen
Xue, Bin
Yang, Ping
Yang, Zijiang
Zavattaro, Gianluigi
Zhan, Naijun

Table of Contents

Session 1. Invited Papers

Session 2. Regular Papers

Session 3. Short Papers

A Process-Theoretic Look at Automata

J.C.M. Baeten, P.J.L. Cuijpers, B. Luttik, and P.J.A. van Tilburg

Division of Computer Science, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands,
{j.c.m.baeten,p.j.l.cuijpers,s.p.luttik,p.j.a.v.tilburg}@tue.nl

Abstract. Automata theory presents roughly three types of automata: finite automata, pushdown automata and Turing machines. The automata are treated as language acceptors, and the expressiveness of the automata models are considered modulo language equivalence. This notion of equivalence is arguably too coarse to satisfactorily deal with a notion of interaction that is fundamental to contemporary computing. In this paper we therefore reconsider the automaton models from automata theory modulo branching bisimilarity, a well-known behavioral equivalence from process theory that has proved to be able to satisfactorily deal with interaction. We investigate to what extent some standard results from automata theory are still valid if branching bisimilarity is adopted as the preferred equivalence.

1 Introduction

Automata theory is the study of abstract computing devices, or "machines" [11]. It presents and studies roughly three types of automata: finite automata, pushdown automata and Turing machines. Finite automata are the simplest kind of automata; they are widely used to model and analyze finite-state state systems. Pushdown automata add to finite automata a restricted kind of unbounded memory in the form of a stack. Turing machines add to finite automata a more powerful notion of memory in the form of an unbounded tape.

In traditional automata theory, automata are treated as language acceptors. The idea is that a string accepted by the automaton represents a particular computation of the automaton, and the language accepted by it thus corresponds with the set of all computations of the automaton. The language-theoretic interpretation of automata is at the basis of all the standard results taught in an undergraduate course on the subject. For instance, the procedure of transforming a nondeterministic finite automaton into a deterministic one is deemed correct because the resulting automaton is language equivalent to the original automaton (two automata are language equivalent if they accept the same language). Another illustrative example is the correspondence between pushdown automata and context-free grammars: for every language generated by a context-free grammar there is a pushdown automaton that accepts it, and vice versa.

The language-theoretic interpretation abstracts from the moments of choice within an automaton. (For instance, it does not distinguish between, on the one

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 1–33, 2010.

hand, the automaton that first accepts an a and subsequently chooses between accepting a b or a c, and, on the other hand, the automaton that starts with a choice between accepting ab and accepting ac.) As a consequence, the language-theoretic interpretation is only suitable under the assumption that an automaton is a stand-alone computational device; it is unsuitable if some form of interaction of the automaton with its environment (user, other automata running in parallel, etc.) may influence the course of computation.

Interaction and nondeterminism nowadays play a crucial role in computing systems. For instance, selecting an option in a web form can lead to different responses and different options in the following form, and so a fixed input string does not look so useful. Also, one browser query will lead to different answers every day, so it is difficult to see a computer as a function from input to output, there is inherent nondeterminism.

Process theory is the study of reactive systems, i.e., systems that depend on interaction with their environment during their execution. In process theory, a system is usually either directly modeled as a labeled transition system (which is actually a generalization of the notion of finite automaton), or as an expression in a process description language with a well-defined operational semantics that assigns a labeled transition system to each expression. In process theory, interaction between systems is treated as a first-class citizen. One its main contributions is a plethora of behavioral equivalences that to more or lesser extent preserve the branching structure of an automaton (see [7] for an overview). One of the finest behavioral equivalences studied in process theory, which arguably preserves all relevant moments of choice in a system, is *branching bisimilarity* [10].

In this paper we shall reconsider some of the standard results from automata theory when automata are considered modulo *branching bisimilarity* instead of language equivalence. We prefer to use branching bisimilarity because it arguably preserves all relevant moments of choice in a system [10]. Note that all the positive results obtained in this paper automatically also hold modulo any of the coarser behavioral equivalences, and hence also modulo Milner's observation equivalence [12]. Furthermore, it is fairly easy to see that most of our negative results also hold modulo observation equivalence; branching structure is needed only to a limited extent.

In Section 3 we consider *regular processes*, defined as branching bisimulation equivalence classes of labeled transition systems associated with finite automata. Most of the results we present in this section are well-known. The section is included for completeness and to illustrate the correspondence between finite automata and a special type of recursive specifications that can be thought of as the process-theoretic counterpart of regular grammars. We will obtain mostly negative results. Naturally, the determinization procedure standardly presented in automata theory to transform a nondeterministic finite automaton into a deterministic one is not valid modulo branching bisimilarity, and not every labeled transition system associated with a finite automaton is described by a regular expression. We do find a process-theoretic variant of the correspondence between finite automata and right-linear grammars, while there is no process-theoretic variant of the correspondence between finite automata and left-linear grammars.

In Section 4 we consider *pushdown processes*, defined as branching bisimulation equivalence classes of labeled transition systems associated with pushdown automata. First we investigate three alternative termination conditions: termination by final state, by empty stack, or by both. Recall that these termination conditions are equivalent from a language-theoretic perspective. We shall prove that, modulo branching bisimilarity, the termination by empty stack and termination by final state interpretations lead to different notions of pushdown process, while the termination by empty stack and the termination by empty stack and final state coincide. We argue that termination by empty stack is better suited for a process-theoretic treatment than termination by final state. Then, we shall investigate the correspondence between pushdown processes (according to the termination by empty stack and final state interpretation) and processes definable by recursive TSP_τ specifications, which can be thought of as the process-theoretic counterpart of context-free grammars. We shall argue that not every pushdown process is definable by a recursive TSP_τ specification and identify a subclass of pushdown processes that are definable by (a special type of) recursive TSP_τ specifications.

In Section 5 we consider *computable processes*, defined as branching bisimulation equivalence classes of labeled transition systems associated with Turing machines. Being a universal model of computation, the Turing machine model has been particularly influential, probably due to its close resemblance with the computer: a Turing machine can be thought of as a computer running a single program that only interacts with the outside world through its memory, and only at the very beginning and the very end of the execution of the program. Thus, the notion of Turing machine focuses on the computational aspect of the execution of a computer; in particular, it abstracts from the interaction that a computer has with its environment (user, other computers in a network, etc.). Since we find interaction important, we shall work with a variation on the notion of Turing machine, known as *off-line* Turing machine. This notion starts with an empty tape, and the machine can take in one input symbol at a time. In [2], a computable process was defined indirectly, by using an encoding of a transition system by means of two computable functions (defined by a standard Turing machine). We show the two ways yield the same set of computable processes.

2 Process Theory

In this section we briefly recap the basic definitions of the process algebra TCP_τ (Theory of Communicating Processes with τ). We refer to [1] for further details.

Syntax. We presuppose a countably infinite *action alphabet* $\mathcal{A}$, and a countably infinite set of *names* $\mathcal{N}$. The actions in $\mathcal{A}$ denote the basic events that a process may perform. In this paper we shall furthermore presuppose a countably infinite *data alphabet* $\mathcal{D}$, a finite set $\mathcal{C}$ of *channels*, and assume that $\mathcal{A}$ includes special actions $c?d$, $c!d$, $c\text{‽}d$ ($d \in \mathcal{D}$, $c \in \mathcal{C}$), which, intuitively, denote the event that datum d is received, sent, or communicated along *channel* c.

Let $\mathcal{N}'$ be a finite subset of $\mathcal{N}$. The set of *process expressions* $\mathcal{P}$ over $\mathcal{A}$ and $\mathcal{N}'$ is generated by the following grammar:

$$p ::= \mathbf{0} \mid \mathbf{1} \mid a.p \mid \tau.p \mid p \cdot p \mid p + p \mid p \parallel p \mid \partial_c(p) \mid \tau_c(p) \mid N \qquad (a \in \mathcal{A}, N \in \mathcal{N}', c \in \mathcal{C})\ .$$

Let us briefly comment on the operators in this syntax. The constant $\mathbf{0}$ denotes *deadlock*, the unsuccessfully terminated process. The constant $\mathbf{1}$ denotes the successfully terminated process. For each action $a \in \mathcal{A}$ there is a unary operator $a.$ denoting action prefix; the process denoted by $a.p$ can do an a-transition to the process denoted by p. The τ-transitions of a process will, in the semantics below, be treated as unobservable, and as such they are the process-theoretic counterparts of the so-called λ- or ϵ-transitions in the theory of automata and formal languages. For convenience, whenever $\mathcal{A}'$ is some subset of $\mathcal{A}$, we write $\mathcal{A}'_\tau$ for $\mathcal{A}' \cup \{\tau\}$. The binary operator $\cdot$ denotes *sequential composition*. The binary operator $+$ denotes *alternative composition* or *choice*. The binary operator $\parallel$ denotes *parallel composition*; actions of both arguments are interleaved, and in addition a communication $c\text{‽}d$ of a datum d on channel c can take place if one argument can do an input action $c?d$ that matches an output action $c!d$ of the other component. The unary operator $\partial_c(p)$ encapsulates the process p in such a way that all input actions $c?d$ and output actions $c!d$ are blocked (for all data) so that communication is enforced. Finally, the unary operator $\tau_c(p)$ denotes abstraction from communication over channel c in p by renaming all communications $c\text{‽}d$ to τ-transitions.

Let $\mathcal{N}'$ be a finite subset of $\mathcal{N}$, used to define processes by means of (recursive) equations. A *recursive specification* E over $\mathcal{N}'$ is a set of equations of the form

$$N \overset{\text{def}}{=} p$$

with as left-hand side a name N and as right-hand side a process expression p. It is required that a recursive specification E contains, for every $N \in \mathcal{N}'$, precisely one equation with N as left-hand side; this equation will be referred to as the *defining equation* for N in $\mathcal{N}'$.

One way to formalize the operational intuitions we have for the syntactic constructions of TCP$_\tau$, is to associate with every process expression a labeled transition system.

Definition 1 (Labelled Transition System). *A* labeled transition system L *is defined as a four-tuple* $(\mathcal{S}, \rightarrow, \uparrow, \downarrow)$ *where:*

1. $\mathcal{S}$ *is a set of states,*
2. $\rightarrow\ \subseteq \mathcal{S} \times \mathcal{A}_\tau \times \mathcal{S}$ *is an* $\mathcal{A}_\tau$*-labeled* transition relation *on* $\mathcal{S}$*,*
3. $\uparrow\ \in \mathcal{S}$ *is the* initial *state,*
4. $\downarrow\ \subseteq \mathcal{S}$ *is the set of* final *states.*

If $(s, a, t) \in \rightarrow$*, we write* $s \xrightarrow{a} t$*. If* s *is a final state, i.e.,* $s \in \downarrow$*, we write* $s\downarrow$*.*

Table 1. Operational rules for a recursive specification E (a ranges over $\mathcal{A}_\tau$, d ranges over $\mathcal{D}$, and c ranges over $\mathcal{C}$)

$$\frac{}{\mathbf{1}\downarrow} \qquad \frac{}{a.p \xrightarrow{a} p}$$

$$\frac{p \xrightarrow{a} p'}{(p+q) \xrightarrow{a} p'} \qquad \frac{q \xrightarrow{a} q'}{(p+q) \xrightarrow{a} q'} \qquad \frac{p\downarrow}{(p+q)\downarrow} \qquad \frac{q\downarrow}{(p+q)\downarrow}$$

$$\frac{p \xrightarrow{a} p'}{p \cdot q \xrightarrow{a} p' \cdot q} \qquad \frac{p\downarrow \quad q \xrightarrow{a} q'}{p \cdot q \xrightarrow{a} q'} \qquad \frac{p\downarrow \quad q\downarrow}{p \cdot q\downarrow}$$

$$\frac{p \xrightarrow{a} p'}{p \parallel q \xrightarrow{a} p' \parallel q} \qquad \frac{q \xrightarrow{a} q'}{p \parallel q \xrightarrow{a} p \parallel q'} \qquad \frac{p\downarrow \quad q\downarrow}{p \parallel q\downarrow}$$

$$\frac{p \xrightarrow{c!d} p' \quad q \xrightarrow{c?d} q'}{p \parallel q \xrightarrow{c\text{‽}d} p' \parallel q'} \qquad \frac{p \xrightarrow{c?d} p' \quad q \xrightarrow{c!d} q'}{p \parallel q \xrightarrow{c\text{‽}d} p' \parallel q'}$$

$$\frac{p \xrightarrow{a} p' \quad a \neq c?d, c!d}{\partial_c(p) \xrightarrow{a} \partial_c(p')} \qquad \frac{p\downarrow}{\partial_c(p)\downarrow}$$

$$\frac{p \xrightarrow{c\text{‽}d} p'}{\tau_c(p) \xrightarrow{\tau} \tau_c(p')} \qquad \frac{p \xrightarrow{a} p' \quad a \neq c\text{‽}d}{\tau_c(p) \xrightarrow{a} \tau_c(p')} \qquad \frac{p\downarrow}{\tau_c(p)\downarrow}$$

$$\frac{p_N \xrightarrow{a} p \quad (N \overset{\text{def}}{=} p_N) \in E}{N \xrightarrow{a} p} \qquad \frac{p_N\downarrow \quad (N \overset{\text{def}}{=} p_N) \in E}{N\downarrow}$$

We use Structural Operational Semantics [15] to associate a transition relation with process expressions: we let $\rightarrow$ be the $\mathcal{A}_\tau$-labeled transition relation induced on the set of process expressions $\mathcal{P}$ by operational rules in Table 1. Note that the operational rules presuppose a recursive specification E.

Let $\rightarrow$ be an $\mathcal{A}_\tau$-labeled transition relation on a set $\mathcal{S}$ of states. For $s, s' \in \mathcal{S}$ and $w \in \mathcal{A}^*$ we write $s \overset{w}{\twoheadrightarrow} s'$ if there exist states $s_0, \ldots, s_n \in \mathcal{S}$ and actions $a_1, \ldots, a_n \in \mathcal{A}_\tau$ such that $s = s_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} s_n = s'$ and w is obtained from $a_1 \cdots a_n$ by omitting all occurrences of τ. If $s \overset{\varepsilon}{\twoheadrightarrow} t$ (ε denotes the empty word), which just means that t is reachable from s by zero or more τ-transitions, then we shall simply write $s \twoheadrightarrow t$.

Definition 2 (Reachability). *A state $t \in \mathcal{S}$ is* reachable *from a state $s \in \mathcal{S}$ if there exists $w \in \mathcal{A}^*$ such that $s \overset{w}{\twoheadrightarrow} t$.*

Definition 3. *Let E be a recursive specification and let p be a process expression. We define the labeled transition system $\mathcal{T}_E(p) = (\mathcal{S}_p, \rightarrow_p, \uparrow_p, \downarrow_p)$ associated with p and E as follows:*

1. *the set of states $\mathcal{S}_p$ consists of all process expressions reachable from p;*
2. *the transition relation $\rightarrow_p$ is the restriction to $\mathcal{S}_p$ of the transition relation $\rightarrow$ defined on all process expressions by the operational rules in Table 1, i.e., $\rightarrow_p = \rightarrow \cap (\mathcal{S}_p \times \mathcal{A}_\tau \times \mathcal{S}_p)$.*
3. *the process expression p is the initial state, i.e. $\uparrow_p = p$; and*
4. *the set of final states consists of all process expressions $q \in \mathcal{S}_p$ such that $q\downarrow$, i.e., $\downarrow_p = \downarrow \cap \mathcal{S}_p$.*

Given the set of (possibly infinite) labeled transition systems, we can divide out different equivalence relations on this set. Dividing out language equivalence throws away too much information, as the moments where choices are made are totally lost, and behavior that does not lead to a final state is ignored. An equivalence relation that keeps all relevant information, and has many good properties, is branching bisimulation as proposed by van Glabbeek and Weijland [10]. For motivations to use branching bisimulation as the preferred notion of equivalence, see [8].

Let $\rightarrow$ be an $\mathcal{A}_\tau$-labeled transition relation, and let $a \in \mathcal{A}_\tau$; we write $s \xrightarrow{(a)} t$ if $s \xrightarrow{a} t$ or $a = \tau$ and $s = t$.

Definition 4 (Branching bisimilarity). *Let $L_1 = (\mathcal{S}_1, \rightarrow_1, \uparrow_1, \downarrow_1)$ and $L_2 = (\mathcal{S}_2, \rightarrow_2, \uparrow_2, \downarrow_2)$ be labeled transition systems. A* branching bisimulation *from L_1 to L_2 is a binary relation $\mathcal{R} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ such that $\uparrow_1 \mathcal{R} \uparrow_2$ and, for all states s_1 and s_2, $s_1 \mathcal{R} s_2$ implies*

1. *if $s_1 \xrightarrow{a}_1 s_1'$, then there exist $s_2', s_2'' \in \mathcal{S}_2$ such that $s_2 \twoheadrightarrow_2 s_2'' \xrightarrow{(a)}_2 s_2'$, $s_1 \mathcal{R} s_2''$ and $s_1' \mathcal{R} s_2'$;*
2. *if $s_2 \xrightarrow{a}_2 s_2'$, then there exist $s_1', s_1'' \in \mathcal{S}_1$ such that $s_1 \twoheadrightarrow_1 s_1'' \xrightarrow{(a)}_1 s_1'$, $s_1'' \mathcal{R} s_2$ and $s_1' \mathcal{R} s_2'$;*
3. *if $s_1\downarrow_1$, then there exists s_2' such that $s_2 \twoheadrightarrow_2 s_2'$ and $s_2'\downarrow_2$; and*
4. *if $s_2\downarrow_2$, then there exists s_1' such that $s_1 \twoheadrightarrow_1 s_1'$ and $s_1'\downarrow_1$.*

The labeled transition systems L_1 and L_2 are branching bisimilar *(notation: $L_1 \leftrightarrows_b L_2$) if there exists a branching bisimulation from L_1 to L_2.*

Branching bisimilarity is an equivalence relation on labeled transition systems [5].

We need as auxiliary notions in our paper the notion of *inert* τ-transition and the notion of *branching degree* of a state. For a definition of these notions we first define the notion of branching bisimulation *on* a labeled transition system, and the notion of *quotient* of a labeled transition system by its maximal branching bisimulation.

Let $L = (\mathcal{S}, \rightarrow, \uparrow, \downarrow)$ be a labeled transition system. A branching bisimulation *on* L is a binary relation $\mathcal{R}$ on $\mathcal{S}$ that satisfies conditions 1–4 of Definition 4 for all s_1 and s_2 such that $s_1 \mathcal{R} s_2$. Let $\mathcal{R}$ be the maximal branching bisimulation on L. Then $\mathcal{R}$ is an equivalence on $\mathcal{S}$; we denote by $[s]_\mathcal{R}$ the equivalence class of $s \in \mathcal{S}$ with respect to $\mathcal{R}$ and by $\mathcal{S}/\mathcal{R}$ the set of all equivalence classes of $\mathcal{S}$ with respect to $\mathcal{R}$. On $\mathcal{S}/\mathcal{R}$ we can define an $\mathcal{A}_\tau$-labeled transition relation $\rightarrow_\mathcal{R}$ by $[s]_\mathcal{R} \xrightarrow{a}_\mathcal{R} [t]_\mathcal{R}$ if, and only if, there exist

$s' \in [s]_{\mathcal{R}}$ and $t' \in [t]_{\mathcal{R}}$ such that $s' \xrightarrow{a} t'$. Furthermore, we define $\uparrow_{\mathcal{R}} = [\uparrow]_{\mathcal{R}}$ and $\downarrow_{\mathcal{R}} = \{s \mid \exists s' \in \downarrow.\ s \in [s']_{\mathcal{R}}\}$. Now, the *quotient* of L by $\mathcal{R}$ is the labeled transition system $L/\mathcal{R} = (\mathcal{S}/\mathcal{R}, \rightarrow/\mathcal{R}, \uparrow/\mathcal{R}, \downarrow/\mathcal{R})$. It is straightforward to prove that each labeled transition system is branching bisimilar to the quotient of this labeled transition system by its maximal branching bisimulation.

Definition 5 (Inert τ-transitions). *Let L be a labeled transition system and let s and t be two states in L. A τ-transition $s \xrightarrow{\tau} t$ is* inert *if s and t are related by the maximal branching bisimulation on L.*

If s and t are distinct states, then an inert τ-transition $s \xrightarrow{\tau} t$ can be *eliminated* from a labeled transition system, e.g., by removing all outgoing transitions of s, changing every outgoing transition $t \xrightarrow{a} u$ from t to an outgoing transition $s \xrightarrow{a} u$, and removing the state t. This operation yields a labeled transition system that is branching bisimilar to the original labeled transition system.

For example, consider Figure 1. Here, the inert τ-transition from state s to t in the transition system on the left is removed by removing the transition $s \xrightarrow{a} u$ and moving all outgoing transitions of t to s, resulting in the transition system on the right. This is possible because s and t are branching bisimilar.

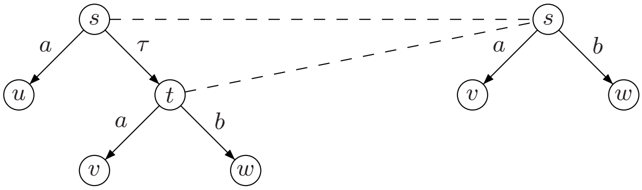


Fig. 1. Removing an inert τ-transition

To get a notion of branching degree that is preserved modulo branching bisimilarity, we define the branching degree of a state as the branching degree of the corresponding equivalence class of states modulo the maximal branching bisimilarity.

Definition 6 (Branching degree). *Let L be a labeled transition system, and let $\mathcal{R}$ be its maximal branching bisimulation. The branching degree of a state s in L is the cardinality of the set $\{(a, [t]_{\mathcal{R}}) \mid [s]_{\mathcal{R}} \xrightarrow{a}_{\mathcal{R}} [t]_{\mathcal{R}}\}$ of outgoing edges of the equivalence class of s in the quotient $L/\mathcal{R}$.*

We say that L has finite branching *if all states of L have a finite branching degree. We say that L has* bounded branching *if there exists a natural number $n \geq 0$ such that every state has a branching degree of at most n.*

Branching bisimulations respect branching degrees in the sense that if $\mathcal{R}$ is a branching bisimulation from L_1 to L_2, s_1 is a state in L_1 and s_2 is a state in L_2 such that $s_1\ \mathcal{R}\ s_2$, then s_1 and s_2 have the same branching degree. Let p and q be process expressions in the context of a recursive specification E; the following properties pertaining to branching degrees are fairly straightforward to establish: If $\mathcal{T}_E(p)$ and $\mathcal{T}_E(q)$ have bounded branching (or finite branching), then $\mathcal{T}_E(p \parallel q)$ has bounded branching (or finite branching) too, and if $\mathcal{T}_E(p)$ has bounded branching (or finite branching), then $\mathcal{T}_E(\partial_c(p))$ and $\mathcal{T}_E(\tau_c(p))$ have bounded branching (or finite branching) too.

3 Regular Processes

A computer with a fixed-size, finite memory is just a finite control. This can be modeled by a finite automaton. Automata theory starts with the notion of a finite automaton. As non-determinism is relevant and basic in concurrency theory, we look at a non-deterministic finite automaton.

Definition 7 (Finite automaton). *A* finite automaton M *is defined as a five-tuple* $(\mathcal{S}, \mathcal{A}', \rightarrow, \uparrow, \downarrow)$ *where:*

1. *$\mathcal{S}$ is a finite set of states,*
2. *$\mathcal{A}'$ is a finite subset of $\mathcal{A}$,*
3. *$\rightarrow \subseteq \mathcal{S} \times \mathcal{A}'_\tau \times \mathcal{S}$ is a finite $\mathcal{A}'_\tau$-labeled* transition relation *on $\mathcal{S}$,*
4. *$\uparrow \in \mathcal{S}$ is the* initial *state,*
5. *$\downarrow \subseteq \mathcal{S}$ is the set of* final *states.*

Clearly, from a finite automaton we obtain a labeled transition system by simply omitting $\mathcal{A}'$ from the five-tuple and declaring $\rightarrow$ to be an $\mathcal{A}_\tau$-labeled transition relation. In the remainder of this paper there is no need to make the formal distinction between a finite automaton and the labeled transition system thus associated to it.

Two examples of finite automata are given in Figure 2.

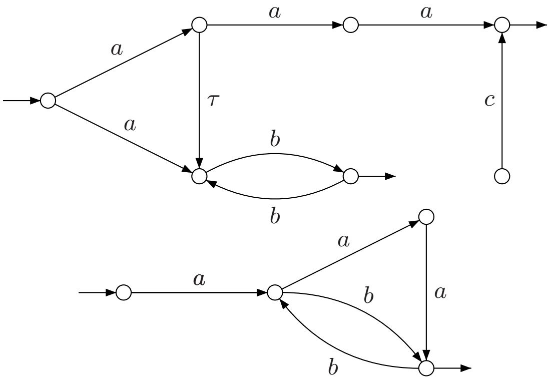


Fig. 2. Two examples of finite automata

Definition 8 (Deterministic finite automaton). *A finite automaton* $M = (\mathcal{S}, \mathcal{A}', \rightarrow, \uparrow, \downarrow)$ *is* deterministic *if, for all states* $s, t_1, t_2 \in \mathcal{S}$ *and for all actions* $a \in \mathcal{A}'$, $s \xrightarrow{a}\!\!\!\!\twoheadrightarrow t_1$ *and* $s \xrightarrow{a}\!\!\!\!\twoheadrightarrow t_2$ *implies* $t_1 = t_2$.

In the theory of automata and formal languages, it is usually also required in the definition of deterministic that the transition relation is *total* in the sense that for all $s \in \mathcal{S}$ and for all $a \in \mathcal{A}'$ there exists $t \in \mathcal{S}$ such that $s \xrightarrow{a} t$. The extra requirement is clearly only sensible in the language interpretation of automata; we shall not be concerned with it here.

The upper automaton in Figure 2 is non-deterministic and has an unreachable c-transition. The lower automaton is deterministic and does not have unreachable transitions; it is not total.

In the theory of automata and formal languages, finite automata are considered as language acceptors.

Definition 9 (Language equivalence). *The* language $\mathcal{L}(L)$ *accepted by a labeled transition system* $L = (\mathcal{S}, \rightarrow, \uparrow, \downarrow)$ *is defined as*

$$\mathcal{L}(L) = \{w \in \mathcal{A}^* \mid \exists s \in \downarrow \text{ such that } \uparrow \xrightarrow{w}\!\!\!\!\twoheadrightarrow s\} \ .$$

Labeled transition systems L_1 *and* L_2 *are* language equivalent *(notation:* $L_1 \equiv L_2$*) if* $\mathcal{L}(L_1) = \mathcal{L}(L_2)$.

Recall that a finite automaton is a special kind of labeled transition system, so the above definition pertains directly to finite automata. The language of both automata in Figure 2 is $\{aaa\} \cup \{ab^{2n-1} \mid n \geq 1\}$; the automata are language equivalent.

A language $L \subseteq \mathcal{A}^*$ accepted by a finite automaton is called a *regular language*. A *regular process* is a branching bisimilarity class of labeled transition systems that contains a finite automaton.

The following standard results pertaining to finite automata are found in every textbook on the theory of automata and formal languages:

1. For every finite automaton there exists a language equivalent automaton without τ-transitions.
2. For every finite automaton there exists a language equivalent *deterministic* finite automaton.
3. Every language accepted by a finite automaton is the language described by a regular expression, and, conversely, every language described by a regular expression is accepted by a finite automaton.
4. Every language accepted by a finite automaton is generated by a regular (i.e., right-linear or left-linear) grammar, and, conversely, every language generated by a regular grammar is accepted by a finite automaton.

We shall discuss below to what extent these results are still valid in branching bisimulation semantics.

Silent steps and non-determinism. Not every regular process has a representation as a finite automaton without τ-transitions, and not every regular process has a representation as a deterministic finite automaton. In fact, it can be proved that there does not exist a finite automaton without τ-transitions that is branching bisimilar with the upper finite automaton in Figure 2. Nor does there exist a deterministic finite automaton branching bisimilar with the upper finite automaton in Figure 2.

Regular grammars and regular expressions. Not every regular process is given by a regular expression, see [3]. We show a simple example in Figure 3 of a finite transition system that is not bisimilar to any transition system that can be associated with a regular expression.

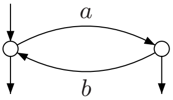


Fig. 3. Not bisimilar to a regular expression

In the theory of automata and formal languages, the notion of *grammar* is used as a syntactic mechanism to describe languages. The corresponding mechanism in concurrency theory is the notion of recursive specification.

We shall now consider the process theory BSP$_\tau$ (Basic Sequential Processes), which is a subtheory of the theory TCP$_\tau$ introduced in Section 2. The syntax of the process theory BSP$_\tau$ is obtained from that of TCP$_\tau$ by omitting sequential composition, parallel composition, encapsulation and abstraction. A BSP$_\tau$ recursive specification over a finite subset $\mathcal{N}'$ of $\mathcal{N}$ is a recursive specification over $\mathcal{N}'$ in which only $\mathbf{0}$, $\mathbf{1}$, N $(N \in \mathcal{N}')$, $a.\_$ $(a \in \mathcal{A}_\tau)$ and $\_+\_$ are used to build process expressions.

Consider the operational rules in Table 1 that are relevant for BSP$_\tau$, for a presupposed recursive specification E. Note that whenever p is a BSP$_\tau$ process expression and $p \xrightarrow{a} q$ then q is again a BSP$_\tau$ process expression. Moreover, q is a subterm of p, or q is a subterm of a right-hand side of the recursive specification E. Thus, it follows that the set of process expressions reachable from a BSP$_\tau$ process expression consists merely of BSP$_\tau$ process expressions, and that it is finite. So the labeled transition system $\mathcal{T}_E(p)$ associated with a BSP$_\tau$ process expression given a BSP$_\tau$ recursive specification E is a finite automaton. Below we shall also establish the converse, that every finite automaton can be specified, up to isomorphism, by a recursive specification over BSP$_\tau$. First we illustrate the construction with an example.

Example 1. Consider the automaton depicted in Figure 4. Note that we have labeled each state of the automaton with a unique name; these will be the names of a recursive specification E. We will define each of these names with an equation, in such a way that the labeled transition system $\mathcal{T}_E(S)$ generated

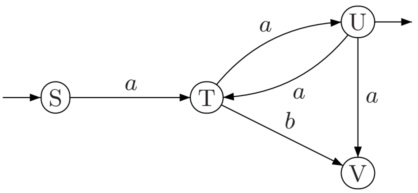


Fig. 4. Example automaton

by the operational semantics in Table 1 is isomorphic (so certainly branching bisimilar) with the automaton in Figure 4.

The recursive specification for the finite automaton in Figure 4 is:

$$\begin{aligned} S &\overset{\text{def}}{=} a.T \ , \\ T &\overset{\text{def}}{=} a.U + b.V \ , \\ U &\overset{\text{def}}{=} a.V + \mathbf{1} \ , \\ V &\overset{\text{def}}{=} \mathbf{0} \ . \end{aligned}$$

The action prefix $a.T$ on the right-hand side of the equation defining S is used to express that S has an a-transition to T. Alternative composition is used on the right-hand side of the defining equation for T to combine the two transitions going out from T. The **1**-summand on the right-hand side of the defining equation for U indicates that U is a final state. The symbol **0** on the right-hand side of the defining equation for V expresses that V is a deadlock state.

Theorem 1. *For every finite automaton M there exists a* BSP_τ *recursive specification E and a* BSP_τ *process expression p such that $M \leftrightarrows_b \mathcal{T}_E(p)$.*

Proof. The general procedure is clear from Example 1. Let $M = (\mathcal{S}, \mathcal{A}', \rightarrow, \uparrow, \downarrow)$. We associate with every state $s \in \mathcal{S}$ a name N_s, and define a recursive specification E on $\{N_s \mid s \in \mathcal{S}\}$. The recursive specification E consists of equations of the form

$$N_s \overset{\text{def}}{=} \sum \{a.N_t \mid s \xrightarrow{a} t\} \ [+\mathbf{1}] \ ,$$

with the contention that the summation $\sum\{a.N_t \mid s \xrightarrow{a} t\}$ denotes **0** if the set $\{a.N_t \mid s \xrightarrow{a} t\}$ is empty, and the optional **1**-summand is present if, and only if, $s\downarrow$. It is easily verified that the binary relation $\mathcal{R} = \{(s, N_s) \mid s \in \mathcal{S}\}$ is a branching bisimulation. □

Incidentally, note that the relation $\mathcal{R}$ in the proof of the above theorem is an isomorphism, so the proof actually establishes that for every finite automaton M there exists a BSP_τ recursive specification E and a BSP_τ process expression p such that the labeled transition system associated with p and E is isomorphic to M.

The above theorem can be viewed as the process-theoretic counterpart of the result from the theory of automata and formal languages that states that every language accepted by a finite automaton is generated by a so-called *right-linear* grammar. There is no reasonable process-theoretic counterpart of the similar result in the theory of automata and formal languages that every language accepted by a finite automaton is generated by a *left-linear* grammar, as we shall now explain.

Table 2. Operational rules for action postfix operators ($a, \beta \in \mathcal{A}_\tau$)

$$\frac{p \xrightarrow{\beta} p'}{p.a \xrightarrow{\beta} p'.a} \qquad \frac{p\downarrow}{p.a \xrightarrow{a} \mathbf{1}}$$

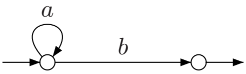


Fig. 5. A simple finite automaton

To obtain the process-theoretic counterpart of a left-linear grammar, we should replace the action prefixes $a.\_$ in BSP$_\tau$ by *action postfixes* $\_.a$, with the operational rules in Table 2. Not every finite automaton can be specified in the resulting language. To see this, note that action postfix distributes over alternative composition and is absorbed by $\mathbf{0}$. Therefore, for every process expression p over BSP$_\tau$ with action postfix instead of action prefix there exist finite sets I and J and elements w_i ($i \in I$) and w_j ($j \in J$) of $\mathcal{A}^*$ such that

$$p \leftrightarrows_{\mathrm{b}} \sum_{i \in I} N_i.w_i + \sum_{j \in J} \mathbf{1}.w_j \; [+\mathbf{1}] \ .$$

(Recall that empty summations are assumed to denote $\mathbf{0}$.) Hence, for every such process expression p, if $p \xrightarrow{a} p'$, then $p' \leftrightarrows_{\mathrm{b}} w$ for some $w \in \mathcal{A}^*$. A process expression denoting the finite automaton in Figure 5 cannot have this property, for after performing an a-transition there is still a choice between terminating with a b-transition, or performing another a-transition. We conclude that the automaton in Figure 5 cannot be described modulo branching bisimilarity in BSP$_\tau$ with action postfix instead of action prefix.

Conversely, with action postfixes instead of action prefixes in the syntax, it is possible to specify labeled transition systems that are not branching bisimilar with a finite automaton.

Example 2. For instance, consider the recursive specification over $\{X\}$ consisting of the equation

$$X \overset{\mathrm{def}}{=} \mathbf{1} + X.a \ .$$

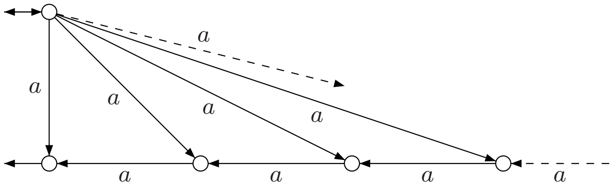


Fig. 6. Infinitely branching process of unguarded equation

The labeled transition system associated with X by the operational semantics is depicted in Figure 6. Note that in this figure, the initial state is also final. It can be proved that the infinitely many states of the depicted labeled transition systems are all distinct modulo branching bisimilarity. It follows that the labeled transition system associated with X is not branching bisimilar to a finite automaton.

We conclude that the classes of processes defined by right-linear and left-linear grammars do not coincide.

4 Pushdown and Context-Free Processes

As an intermediate between the notions of finite automaton and Turing machine, the theory of automata and formal languages treats the notion of pushdown automaton, which is a finite automaton with a stack as memory. Several definitions of the notion appear in the literature, which are all equivalent in the sense that they accept the same languages.

Definition 10 (Pushdown automaton). *A* pushdown automaton *M is defined as a six-tuple $(\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ where:*

1. *$\mathcal{S}$ a finite set of states,*
2. *$\mathcal{A}'$ is a finite subset of $\mathcal{A}$,*
3. *$\mathcal{D}'$ is a finite subset of $\mathcal{D}$,*
4. *$\rightarrow \subseteq \mathcal{S} \times (\mathcal{D}' \cup \{\varepsilon\}) \times \mathcal{A}'_\tau \times \mathcal{D}'^* \times \mathcal{S}$ is a $(\mathcal{D}' \cup \{\varepsilon\}) \times \mathcal{A}'_\tau \times \mathcal{D}'^*$-labeled transition relation on $\mathcal{S}$,*
5. *$\uparrow \in \mathcal{S}$ is the initial state, and*
6. *$\downarrow \subseteq \mathcal{S}$ is the set of final states.*

If $(s, d, a, \delta, t) \in \rightarrow$, we write $s \xrightarrow{d,a,\delta} t$.

The pair of a state together with particular stack contents will be referred to as the *configuration* of a pushdown automaton. Intuitively, a transition $s \xrightarrow{d,a,\delta} t$ (with $a \in \mathcal{A}'$) means that the automaton, when it is in a configuration consisting of a state s and a stack with the datum d on top, can consume input symbol a, replace d by the string δ and move to state t. Likewise, writing $s \xrightarrow{\varepsilon,a,\delta} t$ means that the automaton, when it is in state s and the stack is empty, can consume input symbol a, put the string δ on the stack, and move to state t. Transitions of the form $s \xrightarrow{d,\tau,\delta} t$ or $s \xrightarrow{\varepsilon,\tau,\delta} t$ do not entail the consumption of an input symbol, but just modify the stack contents.

When considering a pushdown automaton as a language acceptor, it is generally assumed that it starts in its initial state with an empty stack. A computation consists of repeatedly consuming input symbols (or just modifying stack contents without consuming input symbols). When it comes to determining whether or not to accept an input string there are two approaches: "acceptance by final state" (FS) and "acceptance by empty stack" (ES). The first approach accepts a string if the pushdown automaton can move to a configuration with a final state by consuming the string, ignoring the contents of the stack in this configuration. The second approach accepts the string if the pushdown automaton can move to a configuration with an empty stack, ignoring whether the state of this configuration is final or not. These approaches are equivalent from a language-theoretic point of view, but not from a process-theoretic point of view, as we shall see below. We shall also consider a third approach in which a configuration is terminating if it consists of a terminating state *and* an empty stack (FSES). We shall see that, from a process-theoretic point of view, the ES and FSES approaches lead to the same notion of pushdown process, whereas the FS approach leads to a different notion.

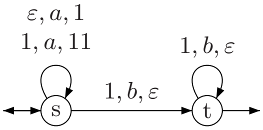


Fig. 7. Example pushdown automaton

Depending on the adopted acceptance condition, the pushdown automaton in Figure 7 accepts the language $\{a^m b^n \mid m \geq n \geq 0\}$ (FS) or the language $\{a^n b^n \mid n \geq 0\}$ (ES,FSES).

Definition 11. *Let $M = (\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ be a pushdown automaton. The labeled transition system $\mathcal{T}(M)$ associated with M is defined as follows:*

1. *the set of states of $\mathcal{T}(M)$ is $\mathcal{S} \times \mathcal{D}'^*$;*
2. *the transition relation of $\mathcal{T}(M)$ satisfies*
   (a) $(s, d\zeta) \xrightarrow{a} (t, \delta\zeta)$ iff $s \xrightarrow{d,a,\delta} t$ for all $s, t \in \mathcal{S}$, $a \in \mathcal{A}'_\tau$, $d \in \mathcal{D}'$, $\delta, \zeta \in \mathcal{D}'^*$, and
   (b) $(s, \varepsilon) \xrightarrow{a} (t, \delta)$ iff $s \xrightarrow{\varepsilon,a,\delta} t$;
3. *the initial state of $\mathcal{T}(M)$ is $(\uparrow, \varepsilon)$; and*
4. *for the set of final states $\downarrow$ we consider three alternative definitions:*
   (a) $(s, \zeta)\downarrow$ in $\mathcal{T}(M)$ iff $s\downarrow$ (the FS interpretation),
   (b) $(s, \zeta)\downarrow$ in $\mathcal{T}(M)$ iff $\zeta = \varepsilon$ (the ES interpretation), and
   (c) $(s, \zeta)\downarrow$ in $\mathcal{T}(M)$ iff $s\downarrow$ and $\zeta = \varepsilon$ (the FSES interpretation).

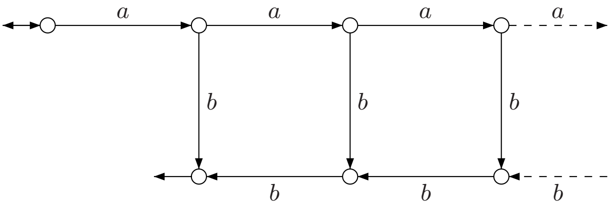


Fig. 8. A pushdown process

This definition now gives us the notions of pushdown language and pushdown process: a *pushdown language* is the language of the transition system associated with a pushdown automaton, and a *pushdown process* is a branching bisimilarity class of labeled transition systems containing a labeled transition system associated with a pushdown automaton.

The labeled transition system in Figure 8 is the labeled transition system associated with the pushdown automaton of Figure 7 according to the FSES or ES interpretations. To obtain the labeled transition system associated to the pushdown automaton in Figure 7 according to the FS interpretation, all states should be made final.

The following standard results pertaining to pushdown automata are often presented in textbooks on the theory of automata and formal languages:

1. Every language accepted by a pushdown automaton under the acceptance by empty stack interpretation is also accepted by a pushdown automaton under the acceptance by final state interpretation, and vice versa.
2. Every language accepted by a pushdown automaton is accepted by a pushdown automaton that only has *push transitions* (i.e., transitions of the form $s \xrightarrow{\varepsilon,a,d} t$ or $s \xrightarrow{d,a,ed}$) and *pop transitions* (i.e., transitions of the form $s \xrightarrow{d,a,\varepsilon} t$).
3. Every language accepted by a pushdown automaton is also generated by a context-free grammar, and every language generated by a context-free grammar is accepted by a pushdown automaton.

Only push and pop transitions. It is easy to see that limiting the set of transitions to push and pop transitions only in the definition of pushdown automaton yields the same notion of pushdown process:

1. Eliminate a transition of the form $s \xrightarrow{\varepsilon,a,\varepsilon} t$ by adding a fresh state s', replacing the transition by the sequence of transitions $s \xrightarrow{\varepsilon,a,d} s' \xrightarrow{d,\tau,\varepsilon} t$ (with d just some arbitrary element in $\mathcal{D}'$).
2. Eliminate a transition of the form $s \xrightarrow{\varepsilon,a,\delta} t$, with $\delta = e_n \cdots e_1$ $(n \geq 1)$, by adding new states $s_2, \ldots, s_n$ and replacing the transition $s \xrightarrow{\varepsilon,a,\delta} t$ by the sequence of transitions
$$s \xrightarrow{\varepsilon,a,e_1} s_2 \xrightarrow{e_1,\tau,e_2e_1} \cdots \xrightarrow{e_{n-2},\tau,e_{n-1}e_{n-2}\cdots e_1} s_n \xrightarrow{e_{n-1},\tau,e_ne_{n-1}\cdots e_1} t \ .$$

3. Eliminate a transition of the form $s \xrightarrow{d,a,\delta} t$, with $\delta = e_n \cdots e_1$ $(n \geq 0)$, by adding new states $s_1, \ldots, s_n$ and replacing the transition $s \xrightarrow{d,a,\delta} t$ by transitions $s \xrightarrow{d,a,\varepsilon} s_1$, $s_1 \xrightarrow{\varepsilon,\tau,e_1} s_2$ and $s_1 \xrightarrow{f,\tau,e_1 f} s_2$ for all $f \in \mathcal{D}'$, and the sequence of transitions

$$s_2 \xrightarrow{e_1,\tau,e_2e_1} \cdots \xrightarrow{e_{n-2},\tau,e_{n-1}e_{n-2}\cdots e_1} s_n \xrightarrow{e_{n-1},\tau,e_ne_{n-1}\cdots e_1} t \ .$$

Branching degree. In [14] the structure of the labeled transition systems associated with pushdown automata was intensively studied (without termination conditions). In particular, they show these labeled transition systems have bounded branching. However, the pushdown processes that are generated modulo branching bisimulation may still exhibit infinite branching. See for example the pushdown automaton in Figure 9 that generates the pushdown process of Figure 6.

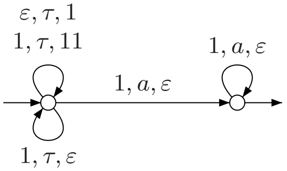


Fig. 9. Pushdown automaton that generates an infinitely branching pushdown process

Nevertheless, we conjecture that whenever a pushdown process has finite branching, it has bounded branching. More precisely:

Conjecture 1. In every pushdown process there is a bound on the branching degree of those states that have finite branching.

Termination conditions. Recall that from a language-theoretic point of view the different approaches to termination of pushdown automata (FS, ES, FSES) are all equivalent, but not from a process-theoretic point of view. First, we argue that the ES and FSES interpretations lead to the same notion of pushdown process.

Theorem 2. *A process is a pushdown process according to the ES interpretation if, and only if, it is a pushdown process according to the FSES interpretation.*

Proof. On the one hand, to see that a pushdown process according to the ES interpretation is also a pushdown process according to the FSES interpretation, let L be the labeled transition system associated with a pushdown automaton M under the ES interpretation, and let M' be the pushdown automaton obtained from M by declaring all states to be final. Then L is the labeled transition system associated with M' under the FSES interpretation.

On the other hand, to see that a pushdown process according to the FSES interpretation is also a pushdown process according to the ES interpretation, let $M = (\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ be an arbitrary pushdown automaton. We shall modify M such that the labeled transition system associated with the modified pushdown automaton under the ES interpretation is branching bisimilar to the labeled transition system associated with M under the FSES interpretation. We define the modified pushdown automaton $M' = (\mathcal{S}', \mathcal{A}', \mathcal{D}' \uplus \{\emptyset\}, \rightarrow', \uparrow', \emptyset)$ as follows:

1. $\mathcal{S}'$ is obtained from $\mathcal{S}$ by adding a fresh initial state $\uparrow'$, and also a fresh state $s^{\downarrow}$ for every final state $s \in \downarrow$;
2. $\rightarrow'$ is obtained from $\rightarrow$ by
   (a) adding a transition $(\uparrow', \varepsilon, \tau, \emptyset, \uparrow)$ (the datum $\emptyset$, which is assumed not to occur in M, is used to mark the end of the stack),
   (b) replacing all transitions $(s, \varepsilon, a, \delta, t) \in \rightarrow$ by $(s, \emptyset, a, \delta\emptyset, t) \in \rightarrow'$, and
   (c) adding transitions $(s, \emptyset, \tau, \varepsilon, s^{\downarrow})$ and $(s^{\downarrow}, \varepsilon, \tau, \emptyset, s)$ for every $s \in \downarrow$.

We leave it to the reader to verify that the relation

$$\mathcal{R} = \{((s, \delta), (s, \delta\emptyset)) \mid s \in \mathcal{S} \ \& \ \delta \in \mathcal{D}'^{*}\} \cup \{((\uparrow, \varepsilon), (\uparrow', \varepsilon))\} \cup \{((s, \varepsilon), (s^{\downarrow}, \varepsilon)) \mid s \in \downarrow\}$$

is a branching bisimulation from the labeled transition associated with M under the ES interpretation to M' under the FSES interpretation. □

If we apply this modification on the pushdown automaton in Figure 7, then we get the result shown in Figure 10 where the states $\uparrow, s^{\downarrow}, t^{\downarrow}$ are added and five transitions, $\uparrow \xrightarrow{\varepsilon,\tau,\emptyset} s$ to put the end-of-stack marker on the stack, $s \xrightarrow{\emptyset,\tau,\varepsilon} s^{\downarrow}$ and $t \xrightarrow{\emptyset,\tau,\varepsilon} t^{\downarrow}$ to remove this marker when in the FSES case termination could occur, and $s^{\downarrow} \xrightarrow{\varepsilon,\tau,\emptyset} s$ and $t^{\downarrow} \xrightarrow{\varepsilon,\tau,\emptyset} t$ to put the end-of-stack marker back.

We proceed to argue that a pushdown process according to the ES interpretation is also a pushdown process according to the FS interpretation, but not vice versa. The classical proof (see, e.g., [11]) that a pushdown language according to the "acceptance by final state" approach is also a pushdown language according to the "acceptance by empty stack" approach coincide employs τ-transitions in a way that is valid modulo language equivalence, but not modulo branching bisimilarity. For instance, the construction that modifies a pushdown automaton

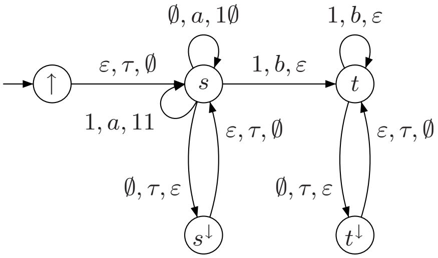

$\emptyset, a, 1\emptyset$
$1, b, \varepsilon$
$\varepsilon, \tau, \emptyset$
$\uparrow$
s
$1, b, \varepsilon$
t
$1, a, 11$
$\varepsilon, \tau, \emptyset$
$\varepsilon, \tau, \emptyset$
$\emptyset, \tau, \varepsilon$
$\emptyset, \tau, \varepsilon$
$s^{\downarrow}$
$t^{\downarrow}$

Fig. 10. Example pushdown automaton accepting on empty stack

M into another pushdown automaton M' such that the language accepted by M by final state is accepted by M' by empty stack adds τ-transitions from every final state of M to a fresh state in M' in which the stack is emptied. The τ-transition introduces, in M', a choice between the original outgoing transitions of the final state in M and termination by going to the fresh state; this choice was not necessarily present in M, and therefore the labeled transition systems associated with M and M' may not be branching bisimilar.

Theorem 3. *A process is a pushdown process according to the ES interpretation only if it is a pushdown process according to the FS interpretation, but not vice versa.*

Proof. On the one hand, to see that a pushdown process according to the ES interpretation is also a pushdown process according to the FS interpretation, let $M = (\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ an arbitrary pushdown automaton. We shall modify M such that the labeled transition system associated with the modified pushdown automaton under the ES interpretation is branching bisimilar to the labeled transition system associated with M under the FS interpretation. We define the modified pushdown automaton $M' = (\mathcal{S}', \mathcal{A}', \mathcal{D}' \uplus \{\emptyset\}, \rightarrow', \uparrow', \downarrow')$ as follows:

1. $\mathcal{S}'$ is obtained from $\mathcal{S}$ by adding a fresh initial state $\uparrow'$, and also a fresh state $s^{\downarrow}$ for every state $s \in \mathcal{S}$;
2. $\downarrow'$ is the set $\{s^{\downarrow} \mid s \in \mathcal{S}\}$ of all these newly added states;
3. $\rightarrow'$ is obtained from $\rightarrow$ by
   (a) adding a transition $(\uparrow', \varepsilon, \tau, \emptyset, \uparrow)$ (the datum $\emptyset$, which is assumed not to occur in M, is used to mark the end of the stack),
   (b) replacing all transitions $(s, \varepsilon, a, \delta, t) \in \rightarrow$ by $(s, \emptyset, a, \delta\emptyset, t) \in \rightarrow'$, and
   (c) adding transitions $(s, \emptyset, \tau, \varepsilon, s^{\downarrow})$ and $(s^{\downarrow}, \varepsilon, \tau, \emptyset, s)$ for every $s \in \mathcal{S}$.

We leave it to the reader to verify that the relation

$$\mathcal{R} = \{((s,\delta),(s,\delta\emptyset)) \mid s \in \mathcal{S} \;\&\; \delta \in \mathcal{D}'^{*}\} \cup \{((\uparrow,\varepsilon),(\uparrow',\varepsilon))\} \cup \{((s,\varepsilon),(s^{\downarrow},\varepsilon)) \mid s \in \mathcal{S}\}$$

is a branching bisimulation from the labeled transition associated with M under the ES interpretation to M' under the FS interpretation.

On the other hand, there exist pushdown processes according to the FS interpretation for which there is no equivalent pushdown process according to the ES interpretation. An example is the pushdown automaton shown in Figure 11.

The labeled transition system associated with it according to the FS interpretation is depicted in Figure 12; it has infinitely many terminating configurations. Moreover, no pair of these configurations is branching bisimilar, which can be seen by noting that the nth state from the left can perform at most $n-1$ times a b-transition before it has to perform an a-transition again.

In contrast with this, note that the labeled transition system associated with any pushdown automaton according to the ES interpretation necessarily has finitely many terminating configurations, for the pushdown automaton has only finitely many states and the stack is required to be empty. □

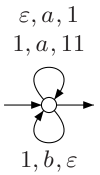


Fig. 11. The counter pushdown automaton

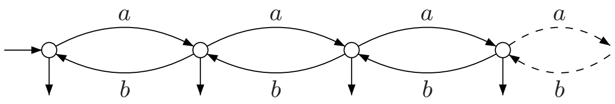


Fig. 12. Labeled transition system associated with automaton of Figure 11 according to the FS interpretation

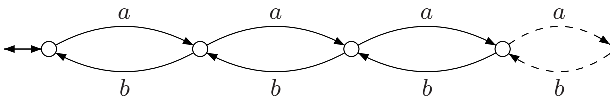


Fig. 13. Labeled transition system associated with automaton of Figure 11 according to the FSES (or ES) interpretation

Context-free specifications. We shall now consider the process-theoretic version of the standard result in the theory of automata and formal languages that the set of pushdown languages coincides with the set of languages generated by context-free grammars. As the process-theoretic counterparts of context-free grammars we shall consider recursive specifications in the subtheory TSP_τ of TCP_τ, which is obtained from BSP_τ by adding sequential composition $\_\cdot\_$. So a TSP_τ recursive specification over a finite subset $\mathcal{N}'$ of $\mathcal{N}$ is a recursive specification over $\mathcal{N}'$ in which only the constructions $\mathbf{0}$, $\mathbf{1}$, N ($N \in \mathcal{N}'$), $a.\_$ ($a \in \mathcal{A}_\tau$), $\_ \cdot \_$ and $\_ + \_$ occur.

TSP_τ recursive specifications can be used to specify pushdown processes. To give an example, the process expression X defined in the TSP_τ recursive specification

$$X \stackrel{\text{def}}{=} \mathbf{1} + a.X \cdot b.\mathbf{1}$$

specifies the labeled transition system in Figure 13, which is associated with the pushdown automaton in Figure 11 under the FSES interpretation.

Next, we will show by contradiction that the FS interpretation of this pushdown automaton (see Figure 12) cannot be given by a TSP_τ recursive specification. Recall that under this interpretation, there are infinitely many distinct states in this pushdown process and all these states are terminating. This implies that all variables in a possible TSP_τ recursive specification for this process would have a **1**-summand to ensure termination in all states. On the other hand, we discuss further on in this paper that any state of a TSP_τ recursive specification can be represented by a sequential composition of variables using the Greibach normal form. Each variable in this normal form must be terminating, since all states are terminating, and each variable can do a bounded number of b-transitions without performing a-transitions in between. To get sequences of b-transitions of arbitrary length, variables are sequentially composed. However, since all variables are also terminating this would result in the possibility to skip parts of the intended sequence of b-transitions and hence lead to branching. This branching is not present in the process in Figure 12, hence this process cannot be represented by a TSP$_\tau$ recursive specification. Since this impossibility already occurs for a very simple example such as a counter, we restrict ourselves to only use the FSES interpretation in the remainder of this paper.

That the notion of TSP$_\tau$ recursive specification still naturally corresponds with with the notion of context-free grammar is confirmed by the following theorem.

Theorem 4. *For every pushdown automaton M there exists a* TSP$_\tau$ *recursive specification E and process expression p such that $\mathcal{T}(M)$ and $\mathcal{T}_E(p)$ are language equivalent, and, vice versa, for every recursive specification E and process expression p there exists a pushdown automaton M such that $\mathcal{T}(M)$ and $\mathcal{T}_E(p)$ are language equivalent.*

We shall see below that a similar result with language equivalence replaced by branching bisimilarity does not hold. In fact, we shall see that there are pushdown processes that are not recursively definable in TSP$_\tau$, and that there are also TSP$_\tau$ recursive specifications that define non-pushdown processes can be defined. We shall present a restriction on pushdown automata and a restriction on TSP$_\tau$ recursive specifications that enable us to retrieve the desired equivalence: we shall prove that the set of so-called *popchoice-free* pushdown processes corresponds with the set of processes definable by a *transparency-restricted* TSP$_\tau$ recursive specification. We have not yet been able to establish that our result is optimal in the sense that a pushdown process is definable by a recursive TSP$_\tau$ specification only if it is popchoice-free, although we conjecture that this is the case.

Consider the pushdown automaton in Figure 14, which generates the transition system shown in Figure 15. In [13], Moller proved that this transition system cannot be defined with a BPA recursive specification, where BPA is the subtheory of TSP$_\tau$ obtained by omitting the τ-prefix and the constant **0** and by disallowing **1** to occur as a summand in a nontrivial alternative composition. His proof can be modified to show that the transition system is not definable with

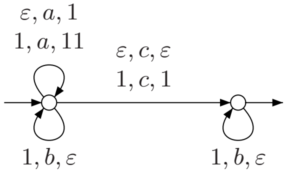


Fig. 14. Pushdown automaton that is not popchoice-free

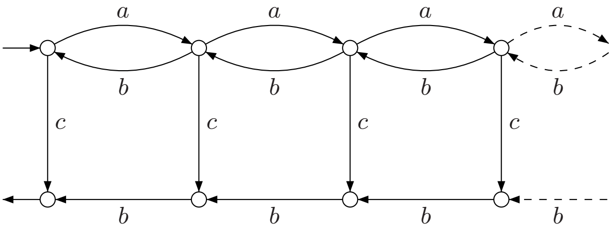


Fig. 15. Transition system of automaton of Figure 14

a TSP_τ recursive specification either. We conclude that not every pushdown process is definable with a TSP_τ recursive specification.

Note that a push of a 1 onto the stack in the initial state of the pushdown automaton in Figure 14 can (on the way to termination) be popped again in the initial state or in the final state: the choice of where the pop will take place cannot be made at the time of the push. In other words, in the pushdown automaton in Figure 14 pop transitions may induce a choice in the associated transition system; we refer to such choice through a pop transition as a *popchoice*. We shall prove below that by disallowing popchoice we define a class of pushdown processes that are definable with a TSP_τ recursive specification.

Definition 12. *Let M be a pushdown automaton that uses only push and pop transitions. A d-pop transition is a transition $s \xrightarrow{d,a,\varepsilon} t$, which pops a datum d. We say M is* popchoice-free *iff whenever there are two d-pop transitions $s \xrightarrow{d,a,\varepsilon} t$ and $s' \xrightarrow{d,b,\varepsilon} t'$, then $t = t'$. A pushdown process is* popchoice-free *if it contains a labeled transition system associated with a popchoice-free pushdown automaton.*

The definition of a pushdown automaton uses a stack as memory. The stack itself can be modeled as a pushdown process, in fact (as we will see shortly) it is the prototypical pushdown process. Given a finite set of data $\mathcal{D}'$, the stack has an input channel i over which it can receive elements of $\mathcal{D}'$ and an output channel o over which it can send elements of $\mathcal{D}'$. The stack process is given by a pushdown automaton with one state $\uparrow$ (which is both initial and final) and transitions $\uparrow \xrightarrow{\varepsilon,i?d,d} \uparrow$, $\uparrow \xrightarrow{e,i?d,de} \uparrow$, and $\uparrow \xrightarrow{d,o!d,\varepsilon} \uparrow$ for all $d, e \in \mathcal{D}'$. As this pushdown automaton has only one state, it is popchoice-free. The transition system of the stack in case $\mathcal{D}' = \{0,1\}$ is presented in Figure 16. The following recursive specification defines a stack:

$$S \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}'} i?d.S \cdot o!d.S \ ; \tag{1}$$

we refer to this specification of a stack over $\mathcal{D}'$ as E_S.

The stack process can be used to make the interaction between control and memory in a pushdown automaton explicit [4]. This is illustrated by the following theorem, stating that every pushdown process is equal to a regular process interacting with a stack.

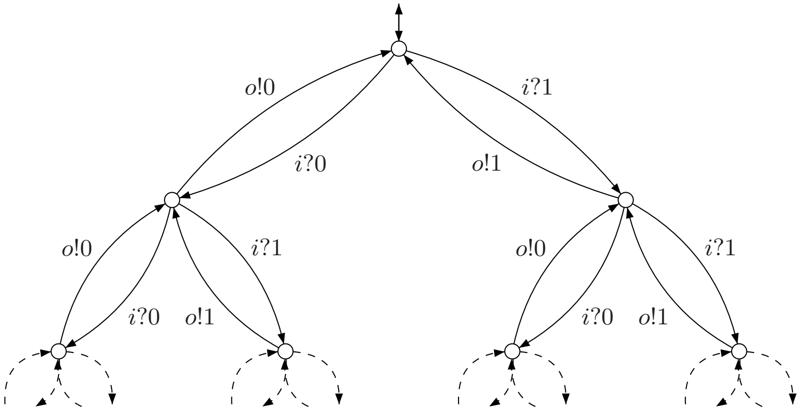


Fig. 16. Stack over $\mathcal{D}' = \{0, 1\}$

Theorem 5. *For every pushdown automaton M there exists a* BSP_τ *process expression p and a* BSP_τ *recursive specification E, and for every* BSP_τ *process expression p and* BSP_τ *recursive specification there exists a pushdown automaton M such that*

$$\mathcal{T}(M) \mathrel{\underline{\leftrightarrow}}_b \mathcal{T}_{E\cup E_S}(\tau_{i,o}(\partial_{i,o}(p \parallel S))) \ .$$

Proof. Let $M = (\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ be a pushdown automaton; we define the BSP_τ recursive specification E as follows:

- For each $s \in \mathcal{S}$ and $d \in \mathcal{D}' \uplus \{\emptyset\}$ it has a variable $V_{s,d}$ (where $\emptyset$ is a special symbol added to $\mathcal{D}'$ to denote that the stack is empty).
- For each pop transition $t \xrightarrow{d,a,\varepsilon} t$ the right-hand side of the defining equation for $V_{s,d}$ has a summand $a.\sum_{e\in\mathcal{D}'\cup\{\emptyset\}} o?e.V_{t,e}$.
- For each push transition $s \xrightarrow{d,a,ed} t$ the right-hand side of the defining equation for $V_{s,d}$ has a summand $a.i!d.V_{t,e}$, and for each push transition $s \xrightarrow{\varepsilon,a,e} t$ the right-hand side of the defining equation for $V_{s,\emptyset}$ has a summand $a.i!\emptyset.V_{t,e}$.
- For each $s \in \mathcal{S}$ such that $s\downarrow$ the right-hand side of the defining equation for $V_{s,\emptyset}$ has a **1**-summand.

We present some observations from which it is fairly straightforward to establish that $\mathcal{T}(M) \mathrel{\underline{\leftrightarrow}}_\mathrm{b} \mathcal{T}_{E\cup E_S}(\tau_{i,o}(\partial_{i,o}(V_{\uparrow,\emptyset} \parallel S)))$. In our proof we abbreviate the process expression $S \cdot i!d_n.S \cdots i!d_1.S$ by $S_{d_n\cdots d_1}$, with, in particular, $S_\varepsilon = S$.

First, note that whenever $\mathcal{T}(M)$ has a transition $(s, d) \xrightarrow{a} (t, \varepsilon)$, then

$$\partial_{i,o}(V_{s,d} \parallel S_\emptyset) \xrightarrow{a} \partial_{i,o}((\sum_{e\in\mathcal{D}'\uplus\{\emptyset\}} o!e.V_{t,e}) \parallel S_\emptyset) \xrightarrow{o?\emptyset} \partial_{i,o}(V_{t,\emptyset} \parallel S) \ .$$

The abstraction operator $\tau_{i,o}(\_)$ will rename the transition labeled $o?\emptyset$ into a τ-transition. This τ-transition is *inert* in the sense that it does not preclude any observable behavior that was possible before the τ-transition. It is well-known that such inert τ-transitions can be omitted while preserving branching bisimilarity.

Second, note that whenever $\mathcal{T}(M)$ has a transition $(s, d\zeta) \xrightarrow{a} (t, \zeta)$ with ζ nonempty, say $\zeta = e\zeta'$, then

$$\partial_{i,o}(V_{s,d} \parallel S_{\zeta}) \xrightarrow{a} \xrightarrow{o?e} \partial_{i,o}(V_{t,e} \parallel S_{\zeta'}) \ ,$$

and, since the second transition is the only step possible after the first a-transition, the τ-transition resulting from applying $\tau_{i,o}(\_)$ is again inert.

Third, note that whenever $\mathcal{T}(M)$ has a transition $(s, d\zeta) \xrightarrow{a} (t, ed\zeta)$, then

$$\partial_{i,o}(V_{s,d} \parallel S_{\zeta}) \xrightarrow{a} \xrightarrow{i?d} \partial_{i,o}(V_{t,e} \parallel S_{d\zeta}) \ ,$$

and again the τ-transition resulting from applying $\tau_{i,o}(\_)$ is inert.

Finally, note that whenever $\mathcal{T}(M)$ has a transition $(s, \varepsilon) \xrightarrow{a} (t, e)$, then

$$\partial_{i,o}(V_{s,\emptyset} \parallel S) \xrightarrow{a} \xrightarrow{i?\emptyset} \partial_{i,o}(V_{t,e} \parallel S_{\emptyset}) \ .$$

Conversely, let E be a BSP$_\tau$ recursive specification, let p be a BSP$_\tau$ process expression, and let $M = (\mathcal{S}, \mathcal{A}', \rightarrow, \uparrow, \downarrow)$ be the associated labeled transition system. We define a pushdown automaton M as follows:

- The set of states, the action alphabet, and the initial and final states are the same as those of the finite automaton.
- The data alphabet is the set of data $\mathcal{D}'$ of the presupposed recursive specification of a stack.
- Whenever $s \xrightarrow{a} t$ in M, and $a \neq i!d, o?d$ $(d \in \mathcal{D}')$, then $s \xrightarrow{d,a,d} t$;
- Whenever $s \xrightarrow{i!d} t$ in M, then $s \xrightarrow{\varepsilon,\tau,d} t$ and $s \xrightarrow{e,\tau,de} t$ for all $e \in \mathcal{D}'$.
- Whenever $s \xrightarrow{o?d} t$ in M, then $s \xrightarrow{d,\tau,\varepsilon} t$.

We omit the proof that every transition of $\mathcal{T}_{E \cup E_S}(\tau_{i,o}(\partial_{i,o}(V_{\uparrow,\emptyset} \parallel S)))$ can be matched by a transition in $\mathcal{T}(M)$ in the sense required by the definition of branching bisimilarity. □

In process theory it is standard practice to restrict attention to *guarded recursive specifications*. Roughly, a TSP$_\tau$ recursive specification is *guarded* if every occurrence of a name occurs in the argument of an action prefix $a.\_$ $(a \in \mathcal{A})$. For a precise definition of guardedness we refer to [1].

Every guarded recursive specification over TSP$_\tau$ can be brought into *restricted Greibach normal form*, that is, satisfying the requirement that every right-hand side of an equation only has summands that are $\mathbf{1}$ or of the form $a.\xi$, where $a \in \mathcal{A}_\tau$ and $\xi = \mathbf{1}$, or ξ is a name, or ξ is a sequential composition of two names. A convenient property of recursive specification in restricted Greibach normal form is that every reachable state in the labeled transition system associated with a name N in such a recursive specification will be denoted by a (generalized) sequential composition of names (see, e.g., the labeled transition system in Figure 17).

Let p be a TSP_τ process expression in the context of a guarded recursive specification E. Then the associated labeled transition system $\mathcal{T}_E(p)$ has finite branching (see, e.g., [1] for a proof). It follows that, e.g., the labeled transition system in Figure 6 is not definable by a guarded recursive specification in restricted Greibach normal form. It is possible with the following unguarded specification:

$$X \overset{\mathrm{def}}{=} \mathbf{1} + X \cdot a.\mathbf{1} \ . \tag{2}$$

This should be contrasted with a standard result in the theory of automata and formal languages that, after translation to our process-theoretic setting, states that even if E is *not* guarded, then still there exists a guarded recursive specification E' in Greibach normal such that $\mathcal{T}_E(p)$ and $\mathcal{T}_{E'}(p)$ are language equivalent.

In this paper we choose to follow the standard practice of using guarded recursive specifications, even though this means that we cannot find a complete correspondence with respect to infinite branching pushdown processes. We leave the generalization of our results to an unguarded setting as future work.

Still, restricting to guarded recursive specifications in restricted Greibach normal form is not sufficient to get the desired correspondence between processes definable by TSP_τ recursive specifications and processes definable as a popchoice-free pushdown automaton. Consider the following guarded recursive specification, which is in restricted Greibach normal form:

$$X \overset{\mathrm{def}}{=} a.X \cdot Y + b.\mathbf{1} \ ,$$
$$Y \overset{\mathrm{def}}{=} \mathbf{1} + c.\mathbf{1} \ .$$

The labeled transition system associated with X, which is depicted in Figure 17, has unbounded branching. So, according to our conjecture, cannot be a pushdown process.

Note that the unbounded branching is due to the **1**-summand in the defining equation for Y by which $Y^n \xrightarrow{c} Y^m$ for all $m < n$. A name N in a recursive

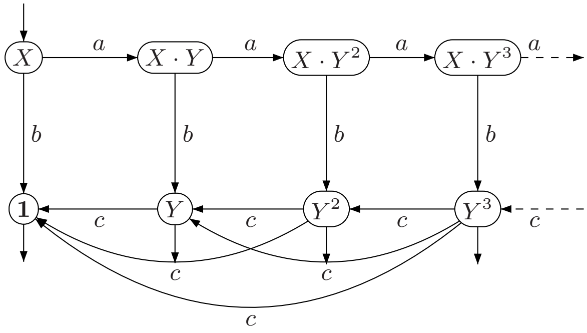


Fig. 17. Process with unbounded branching

specification is called *transparent* if its defining equation has a **1**-summand; otherwise it is called *opaque.* To exclude recursive specifications generating labeled transition systems with unbounded branching, we will require that transparent names may only occur as the *last* element of reachable sequential compositions of names.

Definition 13 (Transparency restricted). *Let E be a recursive specification over* TSP_τ *in restricted Greibach normal form. We call such a specification* transparency-restricted *if for all (generalized) sequential compositions of names ξ reachable from a name in E it holds that all but the last name in ξ is opaque.*

As an example, note that the specification of the stack over $\mathcal{D}'$ defined in (1) above is not transparency restricted, because it is not in Greibach normal form. But the same process can be defined with a transparency-restricted recursive specification: it suffices to add, for all $d \in \mathcal{D}'$, a name T_d to replace $S \cdot o!d.\mathbf{1}$. Thus we obtain the following transparency-restricted specification of the stack over $\mathcal{D}'$:

$$S \overset{\mathrm{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}'} i?d.T_d \cdot S \ ,$$

$$T_d \overset{\mathrm{def}}{=} o!d.\mathbf{1} + \sum_{e \in \mathcal{D}'} i?e.T_e \cdot T_d \ .$$

It can easily be seen that the labeled transition system associated with a name in a transparency-restricted specification has bounded branching: the branching degree of a state denoted by a reachable sequential composition of names is equal to the branching degree of its first name, and the branching degree of a name is bounded by the number of summands of the right-hand side of its defining equation. Since **1**-summands can be eliminated modulo language equivalence (according to the procedure for eliminating λ- or ϵ-productions from context-free grammars), there exists, for *every* TSP_τ recursive specification E a transparency-restricted specification E' such that $\mathcal{T}_E(p)$ and $\mathcal{T}_{E'}(p)$ are *language equivalent* (with p an arbitrary process expression in the context of E).

For investigations under what circumstances we can extend the set of pushdown processes to incorporate processes with unbounded branching, see [4]. In this paper a (partially) forgetful stack is used to deal with transparent variables on the stack. However, if we allow for τ-transitions in the recursive specifications, we can use the stack as is presented above. Note also that the paper does not require the recursive specifications to be transparency-restricted, but this comes at the cost of using a weaker equivalence (namely contrasimulation [9] instead of branching bisimulation) in some cases.

We are now in a position to establish a process-theoretic counterpart of the correspondence between pushdown automata and context-free grammars.

Theorem 6. *A process is a popchoice-free pushdown process if, and only if, it is definable by a transparency-restricted recursive specification.*

Proof. For the implication from right to left, let E be a transparency-restricted recursive specification, and let I be a name in E. We define a pushdown automaton $M = (\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ as follows:

1. The set $\mathcal{S}$ consists of the names occurring in E, the symbol $\mathbf{1}$, an extra initial state $\uparrow$, and an extra intermediate state t.
2. The set $\mathcal{A}'$ consists of all the actions occurring in E.
3. The set $\mathcal{D}'$ consists of the names occurring in E and the symbol $\mathbf{1}$.
4. The transition relation $\rightarrow$ is defined as follows:
   (a) there is a transition $\uparrow \xrightarrow{\varepsilon,\tau,\mathbf{1}} I$;
   (b) if the right-hand side of the defining equation for a name N has a summand $a.\mathbf{1}$, then $\rightarrow$ has transitions $N \xrightarrow{\mathbf{1},a,\varepsilon} \mathbf{1}$ and $N \xrightarrow{N',a,\varepsilon} N'$,
   (c) if the right-hand side of the defining equation for a name N has a summand $a.N'$, then there are transitions $N \xrightarrow{d,a,N'd} t$ and $t \xrightarrow{N',\tau,\varepsilon} N'$ $(d \in \mathcal{D}')$, and
   (d) if the right-hand side of the defining equation for a name N has a summand $a.N' \cdot N''$, then there are transitions $N \xrightarrow{d,a,N''d} N'$ $(d \in \mathcal{D}')$.
5. The set of final states $\downarrow$ consists of $\mathbf{1}$ and all variables with a $\mathbf{1}$-summand.

We leave it to the reader to check that $\mathcal{T}_E(I) \leftrightarroweq_{\mathrm{b}} \mathcal{T}(M)$. Using the procedure described earlier in this section, the set of transitions can be limited to include push and pop transitions only. The pushdown automaton resulting from the procedure is popchoice-free, for an N-pop transition leads to state N.

The proof of the implication from left to right is an adaptation of the classical proof that associates a context-free grammar with a given pushdown automaton. Let $M = (\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ be a popchoice-free pushdown automaton. We define a transparency-restricted specification E with for every state $s \in \mathcal{S}$ a name $N_{s\varepsilon}$ and for every state s a name N_{sdt} if M has transitions that pop datum d leading to the state t. The defining equations in E for these names satisfy the following:

1. The right-hand side of the defining equation for $N_{s\varepsilon}$ has
   (a) a summand $\mathbf{1}$ if, and only if, $s\downarrow$, and
   (b) a summand $a.N_{tdw} \cdot N_{w\varepsilon}$ whenever $s \xrightarrow{\varepsilon,a,d} t$ and all d-pop transitions lead to w.
2. $N_{s\varepsilon} \overset{\mathrm{def}}{=} \mathbf{0}$ if $N_{s\varepsilon}$ has no other summands.
3. The right-hand side of the defining equation for N_{sdt} has
   (a) a summand $a.\mathbf{1}$ if, and only if, $s \xrightarrow{d,a,\varepsilon} t$, and
   (b) a summand $a.N_{uew} \cdot N_{wdt}$ whenever $s \xrightarrow{d,a,ed} u$ and all e-pop transitions lead to state w.
4. $N_{sdt} \overset{\mathrm{def}}{=} \mathbf{0}$ if N_{sdt} has no other summands.

It is easy to see that the resulting specification is transparency-restricted, and that $\mathcal{T}_E(N_{\uparrow\varepsilon}) \leftrightarroweq_{\mathrm{b}} \mathcal{T}(M)$. □

Consider the pushdown automaton shown in Figure 7. It is easy to see that this pushdown automaton is popchoice-free, since both 1-pop transitions lead to the same state t. Using the method described in the proof of Theorem 6 we can now give the following recursive specification over TSP_τ:

$$\begin{aligned}
N_{s\varepsilon} &\stackrel{\text{def}}{=} \mathbf{1} + a.N_{s1t} \cdot N_{t\varepsilon} \ ,\\
N_{t\varepsilon} &\stackrel{\text{def}}{=} \mathbf{1} \ ,\\
N_{s1t} &\stackrel{\text{def}}{=} b.\mathbf{1} + a.N_{s1t} \cdot N_{t1t} \ ,\\
N_{t1t} &\stackrel{\text{def}}{=} b.\mathbf{1} \ .
\end{aligned}$$

We can reduce this specification by removing occurrences of $N_{t\varepsilon}$ (for the right-hand side of the defining equation of this name is just $\mathbf{1}$) and substituting occurrences of N_{t1t} by $b.\mathbf{1}$. We get

$$\begin{aligned}
N_{s\varepsilon} &\stackrel{\text{def}}{=} \mathbf{1} + a.N_{s1t} \ ,\\
N_{s1t} &\stackrel{\text{def}}{=} b.\mathbf{1} + a.N_{s1t} \cdot b.\mathbf{1} \ .
\end{aligned}$$

Now, we see that $N_{s1t} = (\mathbf{1} + a.N_{s1t}) \cdot b.\mathbf{1} = N_{s\varepsilon} \cdot b.\mathbf{1}$ and therefore we have that $N_{s\varepsilon} \stackrel{\text{def}}{=} \mathbf{1} + a.N_{s\varepsilon} \cdot b.\mathbf{1}$ which is equal to the specification we gave before.

Thus, we have established a correspondence between a popchoice-free pushdown processes on the one hand, and transparency-restricted recursive specification over TSP_τ on the other hand, thereby casting the classical result of the equivalence of pushdown automata and context-free grammars in terms of processes and bisimulation.

5 Computable Processes

We proceed to give a definition of a Turing machine that we can use to generate a transition system. The classical definition of a Turing machine uses the memory tape to hold the input string at start up. We cannot use this simplifying trick, as we do not want to fix the input string beforehand, but want to be able to input symbols one symbol at a time. Therefore, we make an adaptation of a so-called *off-line* Turing machine, which starts out with an empty memory tape, and can take an input symbol one at a time. Another important consideration is that we allow termination only when the tape is empty again and we are in a final state: this is like the situation we had for the pushdown automaton.

Definition 14 (Turing machine). *A* Turing machine *M is defined as a six-tuple $(\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ where:*

1. *$\mathcal{S}$ is a finite set of states,*
2. *$\mathcal{A}'$ is a finite subset of $\mathcal{A}$,*
3. *$\mathcal{D}'$ is a finite subset of $\mathcal{D}$,*

4. $\rightarrow \subseteq \mathcal{S} \times (\mathcal{D}' \cup \{\varepsilon\}) \times (\mathcal{A}' \cup \{\tau\}) \times (\mathcal{D}' \cup \{\varepsilon\}) \times \{L, R\} \times \mathcal{S}$ *is a finite set of* transitions *or* steps*,*
5. $\uparrow \in \mathcal{S}$ *is the initial state,*
6. $\downarrow \subseteq \mathcal{S}$ *is the set of final states.*

If $(s, d, a, e, M, t) \in \rightarrow$*, we write* $s \xrightarrow{d,a,e,M} t$*, and this means that the machine, when it is in state* s *and reading symbol* d *on the tape, will execute input action* a*, change the symbol on the tape to* e*, will move one step left if* $M = L$ *and right if* $M = R$ *and thereby move to state* t*. It is also possible that* d *and/or* e *is* ε*: if* d *is* ε*, we are looking at an empty part of the tape, but, if the tape is nonempty, then there is a symbol immediately to the right or to the left; if* e *is* ε*, then a symbol will be erased, but this can only happen at an end of the memory string. The exact definitions are given below.*

At the start of a Turing machine computation, we will assume the Turing machine is in the initial state, and that the memory tape is empty (denoted by $\Box$*).*

By looking at all possible executions, we can define the transition system of a Turing machine. Also Caucal [6] defines the transition system of a Turing machine in this way, but he considers transition systems modulo isomorphism, and leaves out all internal τ-moves.

Definition 15. *Let* $M = (\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ *be a Turing machine. The* labeled transition system *of* M *is defined as follows:*

1. *The set of states is* $\{(s, \bar{\Box}) \mid s \in \mathcal{S}\} \cup \{(s, \Box\delta\Box) \mid s \in \mathcal{S}, \delta \in \mathcal{D}'^{*} - \{\varepsilon\}\}$*, where in the second component there is an overbar on one of the elements of the string* $\Box\delta\Box$ *denoting the contents of the memory tape and the present location. The box indicates a blank portion of the tape.*
2. *A symbol can be replaced by another symbol if the present location is not a blank. Moving right, there are two cases: there is another symbol to the right or there is a blank to the right.*
   - $(s, \Box\delta\bar{d}\Box) \xrightarrow{a} (t, \Box\delta e\bar{\Box})$ *iff* $s \xrightarrow{d,a,e,R} t$ $(d, e \in \mathcal{D}', \delta \in \mathcal{D}'^{*})$,
   - $(s, \Box\delta\bar{d}f\zeta\Box) \xrightarrow{a} (t, \Box\delta e\bar{f}\zeta\Box)$ *iff* $s \xrightarrow{d,a,e,R} t$*, for all* $d, e \in \mathcal{D}', \delta, \zeta \in \mathcal{D}'^{*}$.

   Similarly, there are two cases for a move left.
   - $(s, \Box\bar{d}\delta\Box) \xrightarrow{a} (t, \bar{\Box}e\delta\Box)$ *iff* $s \xrightarrow{d,a,e,L} t$ $(d, e \in \mathcal{D}', \delta \in \mathcal{D}'^{*})$,
   - $(s, \Box\delta f\bar{d}\zeta\Box) \xrightarrow{a} (t, \Box\delta\bar{f}e\zeta\Box)$ *iff* $s \xrightarrow{d,a,e,L} t$*, for all* $d, e \in \mathcal{D}', \delta, \zeta \in \mathcal{D}'^{*}$.
3. *To erase a symbol, it must be at the end of the string. For a move right, there are three cases.*
   - $(s, \Box\bar{d}\Box) \xrightarrow{a} (t, \bar{\Box})$ *iff* $s \xrightarrow{d,a,\varepsilon,R} t$ $(d \in \mathcal{D}')$,
   - $(s, \Box\delta\bar{d}\Box) \xrightarrow{a} (t, \Box\delta\bar{\Box})$ *iff* $s \xrightarrow{d,a,\varepsilon,R} t$ $(d \in \mathcal{D}', \delta \in \mathcal{D}'^{*} - \{\varepsilon\})$,
   - $(s, \Box\bar{d}f\delta\Box) \xrightarrow{a} (t, \Box\bar{f}\delta\Box)$ *iff* $s \xrightarrow{d,a,\varepsilon,R} t$ $(d \in \mathcal{D}', \delta \in \mathcal{D}'^{*})$.

Similarly for a move left.

- $(s, \Box\bar{d}\Box) \xrightarrow{a} (t, \bar{\Box})$ *iff* $s \xrightarrow{d,a,\varepsilon,L} t$ $(d \in \mathcal{D}')$,
- $(s, \Box\bar{d}\delta\Box) \xrightarrow{a} (t, \bar{\Box}\delta\Box)$ *iff* $s \xrightarrow{d,a,\varepsilon,L} t$ $(d \in \mathcal{D}', \delta \in \mathcal{D}'^* - \{\varepsilon\})$,
- $(s, \Box\delta f\bar{d}\Box) \xrightarrow{a} (t, \Box\delta\bar{f}\Box)$ *iff* $s \xrightarrow{d,a,\varepsilon,L} t$ $(d \in \mathcal{D}', \delta \in \mathcal{D}'^*)$.

4. *To insert a new symbol, we must be looking at a blank. We can only move right, if we are to the left of a (possible) data string. This means there are two cases for a move right.*
   - $(s, \bar{\Box}) \xrightarrow{a} (t, \Box d\bar{\Box})$ *iff* $s \xrightarrow{\varepsilon,a,d,R} t$ $(d \in \mathcal{D}')$,
   - $(s, \bar{\Box}f\delta\Box) \xrightarrow{a} (t, \Box d\bar{f}\delta\Box)$ *iff* $s \xrightarrow{\varepsilon,a,d,R} t$ $(d \in \mathcal{D}', \delta \in \mathcal{D}'^*)$.

   Similarly for a move left.
   - $(s, \bar{\Box}) \xrightarrow{a} (t, \bar{\Box}d\Box)$ *iff* $s \xrightarrow{\varepsilon,a,d,L} t$ $(d \in \mathcal{D}')$,
   - $(s, \Box\delta f\bar{\Box}) \xrightarrow{a} (t, \Box\delta\bar{f}\Box)$ *iff* $s \xrightarrow{\varepsilon,a,d,L} t$ $(d \in \mathcal{D}', \delta \in \mathcal{D}'^*)$.
5. *Finally, looking at a blank, we can keep it a blank. Two cases for a move right.*
   - $(s, \bar{\Box}) \xrightarrow{a} (t, \bar{\Box})$ *iff* $s \xrightarrow{\varepsilon,a,\varepsilon,R} t$,
   - $(s, \bar{\Box}f\delta\Box) \xrightarrow{a} (t, \Box\bar{f}\delta\Box)$ *iff* $s \xrightarrow{\varepsilon,a,\varepsilon,R} t$ $(d \in \mathcal{D}', \delta \in \mathcal{D}'^*)$.

   Similarly for a move left.
   - $(s, \bar{\Box}) \xrightarrow{a} (t, \bar{\Box})$ *iff* $s \xrightarrow{\varepsilon,a,\varepsilon,L} t$,
   - $(s, \Box\delta f\bar{\Box}) \xrightarrow{a} (t, \Box\delta\bar{f}\Box)$ *iff* $s \xrightarrow{\varepsilon,a,\varepsilon,L} t$ $(d \in \mathcal{D}', \delta \in \mathcal{D}'^*)$.
6. *The initial state is* $(\uparrow, \bar{\Box})$;
7. $(s, \bar{\Box}) \downarrow$ *iff* $s \downarrow$.

Now we define a *computable process* as the branching bisimulation equivalence class of a transition system of a Turing machine.

In order to make the internal communications of a Turing machine explicit, we need now *two* stacks, one on the left containing the contents of the memory tape to the left of the current symbol and one on the right containing the contents of the memory tape to the right of the current symbol:

$$S^l \overset{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}} li?d.S^l \cdot lo!d.S^l \ ,$$

$$S^r \overset{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}} ri?d.S^r \cdot ro!d.S^r \ .$$

Then, we get the following characterization of computable processes.

Theorem 7. *If process p is a computable process, then there is a regular process q with*

$$p \mathrel{\underline{\leftrightarrow}}_b \tau_{li,lo,ri,ro}(\partial_{li,lo,ri,ro}(q \parallel S^l \parallel S^r)) \ .$$

Proof. Suppose there is a Turing machine $M = (\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ generating a transition system that is branching bisimilar to p. We proceed to define a BSP specification for the regular process q. This specification has variables $V_{s,d}$ for

$s \in \mathcal{S}$ and $d \in \mathcal{D}' \cup \{\emptyset\}$. Moreover, there are variables $W_{s,\emptyset}$ denoting that the tape is empty on both sides.

1. The initial variable is $W_{\uparrow,\emptyset}$;
2. Whenever $s \xrightarrow{d,a,e,r} t$ $(d, e \in \mathcal{D}')$, variable $V_{s,d}$ has a summand
$$a.li!e.\sum_{f \in \mathcal{D}' \cup \{\emptyset\}} ro?f.V_{t,f}$$
3. Whenever $s \xrightarrow{d,a,e,L} t$ $(d, e \in \mathcal{D}')$, variable $V_{s,d}$ has a summand
$$a.ri!e.\sum_{f \in \mathcal{D}' \cup \{\emptyset\}} lo?f.V_{t,f}$$
4. Whenever $s \xrightarrow{d,a,\varepsilon,R} t$ $(d \in \mathcal{D}')$, variable $V_{s,d}$ has a summand
$$a.(ro?\emptyset.(lo?\emptyset.W_{t,\emptyset} + \sum_{f \in \mathcal{D}'} lo?f.li!f.ri!\emptyset.V_{t,\emptyset}) + \sum_{f \in \mathcal{D}'} ro?f.V_{t,f})$$
5. Whenever $s \xrightarrow{d,a,\varepsilon,L} t$ $(d \in \mathcal{D}')$, variable $V_{s,d}$ has a summand
$$a.(lo?\emptyset.(ro?\emptyset.W_{t,\emptyset} + \sum_{f \in \mathcal{D}'} ro?f.ri!f.li!\emptyset.V_{t,\emptyset}) + \sum_{f \in \mathcal{D}'} lo?f.V_{t,f})$$
6. Whenever $s \xrightarrow{\varepsilon,a,d,R} t$, variable $V_{s,\emptyset}$ has a summand
$$a.li!d.(ro?\emptyset.ri!\emptyset.V_{t,\emptyset} + \sum_{f \in \mathcal{D}'} ro?f.V_{t,f})$$
and variable $W_{s,\emptyset}$ has a summand $a.li!\emptyset.li!d.ri!\emptyset.V_{t,\emptyset}$;
7. Whenever $s \xrightarrow{\varepsilon,a,d,L} t$, variable $V_{s,\emptyset}$ has a summand
$$a.ri!d.(lo?\emptyset.li!\emptyset.V_{t,\emptyset} + \sum_{f \in \mathcal{D}'} lo?f.V_{t,f})$$
and variable $W_{s,\emptyset}$ has a summand $a.ri!\emptyset.ri!d.li!\emptyset.V_{t,\emptyset}$;
8. Whenever $s \xrightarrow{\varepsilon,a,\varepsilon,R} t$, variable $V_{s,\emptyset}$ has a summand
$$a.(ro?\emptyset.ri!\emptyset.V_{t,\emptyset} + \sum_{f \in \mathcal{D}'} ro?f.V_{t,f})$$
and variable $W_{s,\emptyset}$ has a summand $a.W_{t,\emptyset}$;
9. Whenever $s \xrightarrow{\varepsilon,a,\varepsilon,L} t$, variable $V_{s,\emptyset}$ has a summand
$$a.(lo?\emptyset.li!\emptyset.V_{t,\emptyset} + \sum_{f \in \mathcal{D}'} lo?f.V_{t,f})$$
and variable $W_{s,\emptyset}$ has a summand $a.W_{t,\emptyset}$;

10. Whenever $s\downarrow$, then variable $W_{s,\emptyset}$ has a summand $\mathbf{1}$.

As before, it can checked that this definition of q satisfies the theorem. □

The converse of this theorem does not hold in full generality, as a regular process can communicate with a pair a stacks in ways that cannot be mimicked by a tape. For instance, by means of the stacks, an extra cell on the tape can be inserted or removed. We can obtain a converse of this theorem, nonetheless, if we interpose, between the regular process and the two stacks, an additional regular process *Tape*, that can only perform actions that relate to tape manipulation, viz.

1. $o!d$ $(d \in \mathcal{D}')$, the current symbol can be read;
2. $o!\varepsilon$, we are looking at a blank cell at the end of the string;
3. $i?e$ $(e \in \mathcal{D}')$, the current symbol can be replaced;
4. $i?\varepsilon$, the current symbol can be erased if we are at an end of the string;
5. $i?L$, a move one cell to the left, executed by pushing the current symbol on top of the right-hand stack and popping the left-hand stack;
6. $i?R$, a move one cell to the right, executed by pushing the current symbol on top of the left-hand stack and popping the right-hand stack.

Thus, we have given a characterization of what is a computable process.

In [2], a computable process was defined in a different way. Starting from a classical Turing machine, the internal communication is made explicit just like we did, by a regular process communicating with two stacks. This shows that a computable function can be described in this way. Next, a computable transition system is coded by means of a computable branching degree function and a computable outgoing edge labeling function. Next, this is again mimicked by a couple of regular processes communicating with a stack. Using this, a similar characterization of computable processes can be reached.

Theorem 8. *A process is computable in the sense of [2] iff it is computable as defined here.*

Proof. Sketch.

If a process is computable in the sense of [2] then we can find a regular process communicating with two stacks such that their parallel composition, after abstraction, is branching bisimilar to it. Moreover, the two stacks together can behave as a tape. Using the theorem above, this means that the process is computable in our sense.

For the other direction, if a process is computable in our sense, then there is a Turing machine for it as defined above. From this Turing machine, we can compute in each state the branching degree and the labels of the outgoing edges. Thus, these functions are computable, and the process is computable in the sense of [2]. □

What remains to be done, is to find a characterization of all recursive specifications over TCP that, after abstraction, give a computable process. In [2], it was found that all guarded recursive specifications over the algebra there yielded computable processes, but that was in the absence of the constant **1**. We already found, in the previous section, that guardedness is not enough in the presence of **1**, and we needed to require transparency-restrictedness. It remains to find a way to extend this notion to all of TCP, so including parallel composition.

6 Conclusion

Every undergraduate curriculum in computer science contains a course on automata theory and formal languages. On the other hand, an introduction to concurrency theory is usually not given in the undergraduate program. Both theories as basic models of computation are part of the foundations of computer science. Automata theory and formal languages provide a model of computation where interaction is not taken into account, so a computer is considered as a stand-alone device executing batch processes. On the other hand, concurrency theory provides a model of computation where interaction is taken into account. Concurrency theory is sometimes called the theory of reactive processes.

Both theories can be integrated into one course in the undergraduate curriculum, providing students with the foundation of computing. This paper provides a glimpse of what happens to the Chomsky hierarchy in a concurrency setting, taking a labeled transition system as a central notion, and dividing out bisimulation semantics on such transition systems.

References

1. Baeten, J.C.M., Basten, T., Reniers, M.A.: Process Algebra (Equational Theories of Communicating Processes). Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge (2009)
2. Baeten, J.C.M., Bergstra, J.A., Klop, J.W.: On the consistency of Koomen's fair abstraction rule. Theoretical Computer Science 51, 129–176 (1987)
3. Baeten, J.C.M., Corradini, F., Grabmayer, C.A.: A characterization of regular expressions under bisimulation. Journal of the ACM 54(2), 1–28 (2007)
4. Baeten, J.C.M., Cuijpers, P.J.L., van Tilburg, P.J.A.: A context-free process as a pushdown automaton. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 98–113. Springer, Heidelberg (2008)
5. Basten, T.: Branching bisimilarity is an equivalence indeed!. Information Processing Letters 58(3), 141–147 (1996)
6. Caucal, D.: On the transition graphs of Turing machines. In: Margenstern, M., Rogozhin, Y. (eds.) MCU 2001. LNCS, vol. 2055, pp. 177–189. Springer, Heidelberg (2001)
7. van Glabbeek, R.J.: The Linear Time – Branching Time Spectrum II. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 66–81. Springer, Heidelberg (1993)
8. van Glabbeek, R.J.: What is Branching Time Semantics and why to use it?. Bulletin of the EATCS 53, 190–198 (1994)
9. van Glabbeek, R.J.: The Linear Time – Branching Time Spectrum I. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, pp. 3–99. Elsevier, Amsterdam (2001)

10. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. Journal of the ACM 43(3), 555–600 (1996)
11. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Pearson, London (2006)
12. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
13. Moller, F.: Infinite results. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 195–216. Springer, Heidelberg (1996)
14. Muller, D.E., Schupp, P.E.: The theory of ends, pushdown automata, and second-order logic. Theoretical Computer Science 37, 51–75 (1985)
15. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)

Verification, Performance Analysis and Controller Synthesis for Real-Time Systems

Uli Fahrenberg, Kim G. Larsen*, and Claus R. Thrane

Department of Computer Science, Aalborg University, Denmark
kgl@cs.aau.dk

Abstract. This article aims at providing a concise and precise *Travellers Guide*, *Phrase Book* or *Reference Manual* to the timed automata modeling formalism introduced by Alur and Dill [7, 8]. The paper gives comprehensive definitions of timed automata, priced (or weighted) timed automata, and timed games and highlights a number of results on associated decision problems related to model checking, equivalence checking, optimal scheduling, and the existence of winning strategies.

1 Introduction

The model of timed automata, introduced by Alur and Dill [7, 8], has by now established itself as a classical formalism for describing the behaviour of real-time systems. A number of important algorithmic problems has been shown decidable for it, including reachability, model checking and several behavioural equivalences and preorders.

By now, real-time model checking tools such as UppAal [17,57] and Kronos [32] are based on the timed automata formalism and on the substantial body of research on this model that has been targeted towards transforming the early results into practically efficient algorithms — *e.g.* [13, 14, 19, 21] — and data structures — *e.g.* [20, 54, 56].

The maturity of a tool like UppAal is witnessed by the numerous applications — *e.g.* [38, 40, 45, 48, 52, 55, 60, 61] — to the verification of industrial case-studies spanning real-time controllers and real-time communication protocols. More recently, model-checking tools in general and UppAal in particular have been applied to solve realistic scheduling problems by a reformulation as reachability problems — *e.g.* [1, 42, 47, 62].

Aiming at providing methods for performance analysis, a recent extension of timed automata is that of *priced* or *weighted* timed automata [9,18], which makes it possible to formulate and solve *optimal* scheduling problems. Surprisingly, a number of properties have been shown to be decidable for this formalism [9,18,29,41,58]. The recently developed UppAal Cora tool provides an efficient tool for solving cost-optimal reachability problems [53] and has been applied successfully to a number of optimal scheduling problems, *e.g.* [15, 22, 44].

* Corresponding author.

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 34–61, 2010.

Most recently, substantial efforts have been made on the automatic synthesis of (correct-by-construction) controllers from timed games for given control objectives. From early decidability results [12, 63] the effort has lead to efficient on-the-fly algorithms [34, 68] with the newest of the UppAal toolset, UppAal Tiga [16], providing an efficient tool implementation with industrial applications emerging, *e.g.* [50].

This survey paper aims at providing a concise and precise *Travellers Guide*, *Phrase Book* or *Reference Manual* to the land and language of timed automata. The article gives comprehensive definitions of timed automata, weighted timed automata, and timed games and highlights a number of results on associated decision problems related to model checking, equivalence checking, optimal scheduling, and the existence of winning strategies. The intention is that the paper should provide an easy-to-access collection of important results and overview of the field to anyone interested.

The authors are indebted to an anonymous reviewer who provided a number of useful comments for improving the paper.

2 Timed Automata

In this section we review the notion of timed automata introduced by Alur and Dill [7, 8] as a formalism for describing the behaviour of real-time systems. We review the syntax and semantics and highlight the, by now classical, region construction underlying the decidability of several associated problems.

Here we illustrate how regions are applied in showing decidability of reachability and timed and untimed (bi)similarity. However, though indispensable for showing decidability, the notion of region does not provide the means for efficient tool implementations. The verification engine of UppAal instead applies so-called zones, which are *convex unions* of regions. We give a brief account of zones as well as their efficient representation and manipulation using difference-bound matrices.

2.1 Syntax and Semantics

Definition 1. *The set $\Phi(C)$ of* clock constraints *φ over a finite set (of* clocks*) C is defined by the grammar*

$$\varphi ::= x \bowtie k \mid \varphi_1 \wedge \varphi_2 \qquad (x \in C, k \in \mathbb{Z}, \bowtie \in \{\leq, <, \geq, >\}).$$

The set $\Phi^+(C)$ of extended clock constraints *φ is defined by the grammar*

$$\varphi ::= x \bowtie k \mid x - y \bowtie k \mid \varphi_1 \wedge \varphi_2 \qquad (x, y \in C, k \in \mathbb{Z}, \bowtie \in \{\leq, <, \geq, >\}).$$

Remark 1. The clock constraints in $\Phi(C)$ above are also called *diagonal-free* clock constraints, and the additional ones in $\Phi^+(C)$ are called *diagonal.* We restrict ourselves to diagonal-free clock constraints here; see Remark 4 for one reason. For additional modelling power, timed automata with diagonal constraints can be used, as it is shown in [8,26] that any such automaton can be converted to a diagonal-free one; however the conversion may lead to an exponential blow-up.

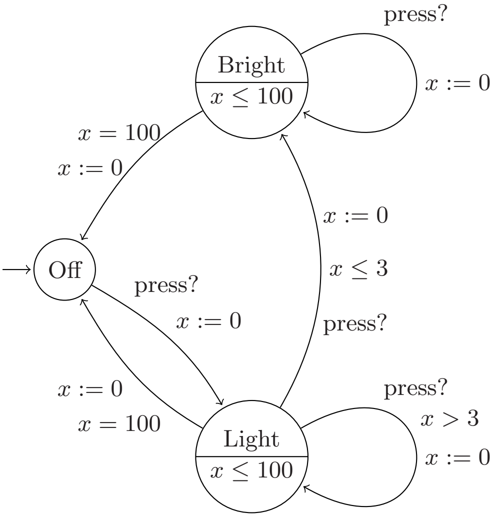


Fig. 1. A light switch modelled as a timed automaton

Definition 2. *A* timed automaton *is a tuple* $(L, \ell_0, F, C, \Sigma, I, E)$ *consisting of a finite set L of locations, an initial location* $\ell_0 \in Q$*, a set* $F \subseteq Q$ *of final locations, a finite set* C *of clocks, a finite set* Σ *of actions, a location invariants mapping* $I : L \to \Phi(C)$*, and a set* $E \subseteq L \times \Phi(C) \times \Sigma \times 2^C \times L$ *of edges.*

We shall denote an edge $(\ell, \varphi, a, r, \ell') \in E$ by $\ell \xrightarrow{\varphi,a,r} \ell'$. Also, 2^C denotes the set of all subsets of C; in general, we will write B^A for the set of mappings from a set A to a set B.

Example 1. Figure 1 provides a timed automaton model of an intelligent light switch. Starting in the "Off" state, a press of the button turns the light on, and it remains in this state for 100 time units (*i.e.* until clock $x = 100$), at which time the light turns off again. During this time, an additional press resets the clock x and prolongs the time in the state by 100 time units. Pressing the button twice, with at most three time units between the presses, triggers a special bright light.

Definition 3. *A* clock valuation *on a finite set* C *of clocks is a mapping* $v : C \to \mathbb{R}_{\geq 0}$*. The* initial *valuation* v_0 *is given by* $v_0(x) = 0$ *for all* $x \in C$*. For a valuation* v*,* $d \in \mathbb{R}_{\geq 0}$*, and* $r \subseteq C$*, the valuations* $v + d$ *and* $v[r]$ *are defined by*

$$(v + d)(x) = v(x) + d$$

$$v[r](x) = \begin{cases} 0 & \text{for } x \in r, \\ v(x) & \text{for } x \notin r. \end{cases}$$

Definition 4. *The* zone *of an extended clock constraint in $\Phi^+(C)$ is a set of clock valuations $C \to \mathbb{R}_{\geq 0}$ given inductively by*

$$\begin{aligned} [\![x \bowtie k]\!] &= \{v : C \to \mathbb{R}_{\geq 0} \mid v(x) \bowtie k\}, \\ [\![x - y \bowtie k]\!] &= \{v : C \to \mathbb{R}_{\geq 0} \mid v(x) - v(y) \bowtie k\}, \text{ and} \\ [\![\varphi_1 \wedge \varphi_2]\!] &= [\![\varphi_1]\!] \cap [\![\varphi_2]\!]. \end{aligned}$$

We shall write $v \models \varphi$ instead of $v \in [\![\varphi]\!]$.

Definition 5. *The* semantics *of a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$ is the transition system $[\![A]\!] = (S, s_0, \Sigma \cup \mathbb{R}_{\geq 0}, T = T_s \cup T_d)$ given as follows:*

$$\begin{aligned} S &= \big\{(\ell, v) \in L \times \mathbb{R}_{\geq 0}^C \bigm| v \models I(\ell)\big\} \qquad\qquad s_0 = (\ell_0, v_0) \\ T_s &= \big\{(\ell, v) \xrightarrow{a} (\ell', v') \bigm| \exists \ell \xrightarrow{\varphi, a, r} \ell' \in E : v \models \varphi, v' = v[r]\big\} \\ T_d &= \big\{(\ell, v) \xrightarrow{d} (\ell, v + d) \bigm| \forall d' \in [0, d] : v + d' \models I(\ell)\big\} \end{aligned}$$

Remark 2. The transition system $[\![A]\!]$ from above is an example of what is known as a *timed transition system*, *i.e.* a transition system where the label set includes $\mathbb{R}_{\geq 0}$ as a subset and which satisfies certain additivity and time determinacy properties. We refer to [2] for a more in-depth treatment.

Also note that the semantics $[\![A]\!]$ contains no information about final states (derived from the final locations in F); this is mostly for notational convenience.

Definition 6. *A (finite)* run *of a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$ is a finite path $\rho = (\ell_0, v_0) \to \cdots \to (\ell_k, v_k)$ in $[\![A]\!]$. It is said to be* accepting *if $\ell_k \in F$.*

Example 1 (continued). The light switch model from figure 1 has as state set

$$S = \{\text{Off}\} \times \mathbb{R}_{\geq 0} \cup \{\text{Light}, \text{Bright}\} \times [0, 100]$$

where we identify valuations with their values at x. A few example runs are given below; we abbreviate "press?" to "p":

$$\begin{aligned} &(\text{Off}, 0) \xrightarrow{150} (\text{Off}, 150) \xrightarrow{\text{p}} (\text{Light}, 0) \xrightarrow{100} (\text{Light}, 100) \to (\text{Off}, 0) \\ &(\text{Off}, 0) \xrightarrow{\text{p}} (\text{Light}, 0) \xrightarrow{10} (\text{Light}, 10) \xrightarrow{\text{p}} (\text{Light}, 0) \xrightarrow{100} (\text{Light}, 100) \to (\text{Off}, 0) \\ &(\text{Off}, 0) \xrightarrow{\text{p}} (\text{Light}, 0) \xrightarrow{1} (\text{Light}, 1) \xrightarrow{\text{p}} (\text{Bright}, 0) \xrightarrow{100} (\text{Bright}, 100) \to (\text{Off}, 0) \end{aligned}$$

2.2 Reachability

We are concerned with the following problem: Given a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$, is any of the locations in F reachable? We shall later define the *timed language* generated by a timed automaton and see that this reachability problem is equivalent to *emptiness checking*: Is the timed language generated by A non-empty?

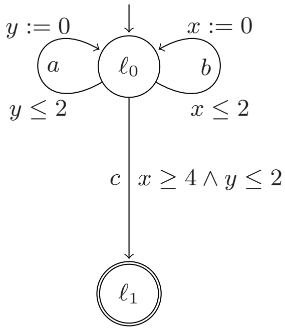


Fig. 2. A timed automaton with two clocks

Example 2 (cf. [2, Ex. 11.7]). Figure 2 shows a timed automaton A with two clocks and a final location ℓ_1. To ask whether ℓ_1 is reachable amounts for this automaton to the question whether there is a finite sequence of a- and b-transitions from ℓ_0 which brings clock values into accordance with the guard $x \geq 4 \wedge y \leq 2$ on the edge leading to ℓ_1.

An immediate obstacle to reachability checking is the infinity of the state space of A. In general, the transition system $[\![A]\!]$ has uncountably many states, hence straight-forward reachability algorithms do not work for us.

Notation 1. *The* derived transition relations *in a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$ are defined as follows: For (ℓ, v), (ℓ', v') states in $[\![A]\!]$, we say that*

- *$(\ell, v) \xrightarrow{\delta} (\ell', v')$ if $(\ell, v) \xrightarrow{d} (\ell', v')$ in $[\![A]\!]$ for some $d > 0$,*
- *$(\ell, v) \xrightarrow{\alpha} (\ell', v')$ if $(\ell, v) \xrightarrow{a} (\ell', v')$ in $[\![A]\!]$ for some $a \in \Sigma$, and*
- *$(\ell, v) \rightsquigarrow (\ell', v')$ if $(\ell, v) (\xrightarrow{\delta} \cup \xrightarrow{\alpha})^* (\ell', v')$.*

Definition 7. *The set of* reachable locations *in a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$ is*

$$\text{Reach}(A) = \big\{ \ell \in L \;\big|\; \exists v : C \to \mathbb{R}_{\geq 0} : (\ell_0, v_0) \rightsquigarrow (\ell, v) \big\}.$$

Hence we can now state the reachability problem as follows:

Problem 1 (Reachability). Given a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$, is $\text{Reach}(A) \cap F \neq \emptyset$?

Definition 8. *Let $A = (L, \ell_0, F, C, \Sigma, I, E)$ be a timed automaton. A reflexive, transitive relation $R \subseteq L \times \mathbb{R}_{\geq 0}^C \times L \times \mathbb{R}_{\geq 0}^C$ is a* time-abstracted simulation *provided that for all $(\ell_1, v_1) \mathrel{R} (\ell_2, v_2)$,*

- *for all $(\ell_1, v_1) \xrightarrow{\delta} (\ell_1', v_1')$ there exists some (ℓ_2', v_2') such that $(\ell_1', v_1') \mathrel{R} (\ell_2', v_2')$ and $(\ell_2, v_2) \xrightarrow{\delta} (\ell_2', v_2')$, and*

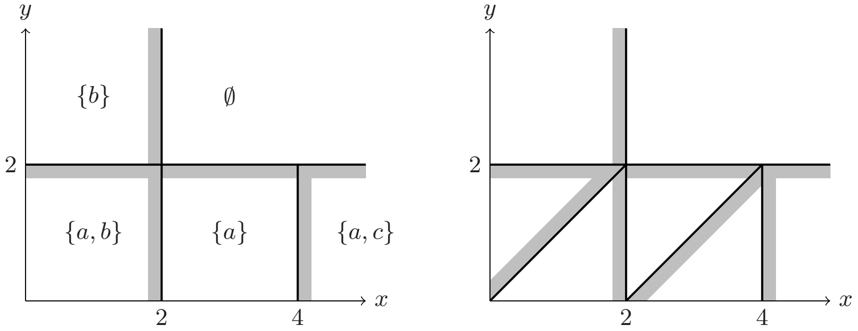


Fig. 3. Time-abstracted bisimulation classes for the two-clock timed automaton from Example 2. Left: equivalence classes for switch transitions only; right: equivalence classes for switch and delay transitions.

- *for all $a \in \Sigma$ and $(\ell_1, v_1) \xrightarrow{a} (\ell_1', v_1')$, there exists some (ℓ_2', v_2') such that $(\ell_1', v_1') \; R \; (\ell_2', v_2')$ and $(\ell_2, v_2) \xrightarrow{a} (\ell_2', v_2')$.*

The relation R is called a time-abstracted bisimulation *if it is also symmetric; it is said to be F-*sensitive *if additionally, $(\ell_1, v_1) \; R \; (\ell_2, v_2)$ implies that $\ell_1 \in F$ if and only if $\ell_2 \in F$.*

Note that for ease of exposition, we require (bi)simulation relations to be reflexive and transitive here; hence bisimulations are equivalence relations. Except for this, a time-abstracted (bi)simulation on A is the same as a standard (bi)simulation on the transition system derived from $[\![A]\!]$ with transitions $\xrightarrow{\delta}$ and $\xrightarrow{a}$. Likewise, the quotient introduced below is just the bisimulation quotient of that derived transition system.

Definition 9. *Let $A = (L, \ell_0, F, C, \Sigma, I, E)$ be a timed automaton and $R \subseteq L \times \mathbb{R}_{\geq 0}^C \times L \times \mathbb{R}_{\geq 0}^C$ a time-abstracted bisimulation. The* quotient *of $[\![A]\!] = (S, s_0, \Sigma \cup \mathbb{R}_{\geq 0}, T)$ with respect to R is the transition system $[\![A]\!]_R = (S_R, s_R^0, \Sigma \cup \{\delta\}, T_R)$ given by $S_R = S/R$, $s_R^0 = [s_0]_R$, and with transitions*

- *$\pi \xrightarrow{\delta} \pi'$ whenever $(\ell, v) \xrightarrow{\delta} (\ell', v')$ for some $(\ell, v) \in \pi$, $(\ell', v') \in \pi'$, and*
- *$\pi \xrightarrow{a} \pi'$ whenever $(\ell, v) \xrightarrow{a} (\ell', v')$ for some $(\ell, v) \in \pi$, $(\ell', v') \in \pi'$.*

The following proposition expresses that F-sensitive quotients are sound and complete with respect to reachability.

Proposition 1 ([4]). *Let $A = (L, \ell_0, F, C, \Sigma, I, E)$ be a timed automaton, $R \subseteq L \times \mathbb{R}_{\geq 0}^C \times L \times \mathbb{R}_{\geq 0}^C$ an F-sensitive time-abstracted bisimulation and $\ell \in F$. Then $\ell \in \text{Reach}(A)$ if and only if there is a reachable state π in $[\![A]\!]_R$ and $v : C \to \mathbb{R}_{\geq 0}$ such that $(\ell, v) \in \pi$.*

Example 2 (continued). We shall now try to construct, in a naïve way, a time-abstracted bisimulation R for the timed automaton A from Figure 2 which is as coarse as possible. Note first that we cannot have $(\ell_0, v)\ R\ (\ell_1, v')$ for any $v, v' : C \to \mathbb{R}_{\geq 0}$ because $\ell_1 \in F$ and $\ell_0 \notin F$. On the other hand it is easy to see that we can let $(\ell_1, v)\ R\ (\ell_1, v')$ for all $v, v' : C \to \mathbb{R}_{\geq 0}$, which leaves us with constructing R on the states involving ℓ_0.

We handle switch transitions $\xrightarrow{\alpha}$ first: If $v, v' : C \to \mathbb{R}_{\geq 0}$ are such that $v(y) \leq 2$ and $v'(y) > 2$, the state (ℓ_0, v) has an a-transition available while the state (ℓ_0, v') has not, hence these cannot be related in R. Similarly we have to distinguish states (ℓ_0, v) from states (ℓ_0, v') where $v(x) \leq 2$ and $v'(x) > 2$ because of b-transitions, and states (ℓ_0, v) from states (ℓ_0, v') where $v(x) < 4$ and $v'(x) \geq 4$ because of c-transitions. Altogether this gives the five classes depicted to the left of Figure 3, where the shading indicates to which class the boundary belongs, and we have written the set of available actions in the classes.

When also taking delay transitions $\xrightarrow{\delta}$ into account, one has to partition the state space further: From a valuation v in the class marked $\{a, b\}$ in the left of the figure, a valuation in the class marked $\{a\}$ can only be reached by a delay transition if $v(y) < v(x)$; likewise, from the $\{a\}$ class, the $\{a, c\}$ class can only be reached if $v(y) \leq v(x) - 2$. Hence these two classes need to be partitioned as shown to the right of Figure 3.

It can easily be shown that no further partitioning is needed, thus we have defined the coarsest time-abstracted bisimulation relation for A, altogether with eight equivalence classes.

2.3 Regions

Motivated by the construction in the example above, we now introduce a time-abstracted bisimulation with a *finite quotient.* To ensure finiteness, we need the maximal constants to which respective clocks are compared in the invariants and guards of a given timed automaton. These may be defined as follows.

Definition 10. *For a finite set C of clocks, the* maximal constant *mapping $c_{\max} : C \to \mathbb{Z}^{\Phi(C)}$ is defined inductively as follows:*

$$c_{\max}(x)(y \bowtie k) = \begin{cases} k & \textit{if } y = x \\ 0 & \textit{if } y \neq x \end{cases}$$

$$c_{\max}(x)(\varphi_1 \wedge \varphi_2) = \max\big(c(x)(\varphi_1), c(x)(\varphi_2)\big)$$

For a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$, the maximal constant mapping is $c_A : C \to \mathbb{Z}$ defined by

$$c_A(x) = \max\big\{c_{\max}(x)(I(\ell)), c_{\max}(x)(\varphi) \bigm| \ell \in L, \ell \xrightarrow{\varphi, a, r} \ell' \in E\big\}.$$

Notation 2. *For $d \in \mathbb{R}_{\geq 0}$ we write $\lfloor d \rfloor$ and $\langle d \rangle$ for the integral, respectively fractional, part of d, so that $d = \lfloor d \rfloor + \langle d \rangle$.*

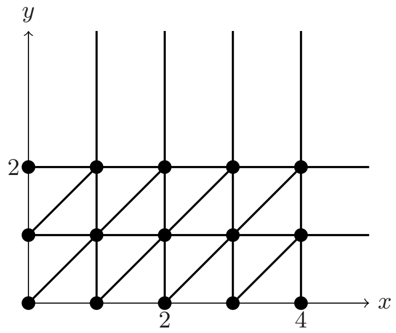


Fig. 4. Clock regions for the timed automaton from Example 2

Definition 11. *For a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$, valuations $v, v' : C \to \mathbb{R}_{\geq 0}$ are said to be* region equivalent*, denoted $v \cong v'$, if*

- $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ *or* $v(x), v'(x) > c_A(x)$*, for all* $x \in C$*, and*
- $\langle v(x) \rangle = 0$ *iff* $\langle v'(x) \rangle = 0$*, for all* $x \in C$*, and*
- $\langle v(x) \rangle \leq \langle v(y) \rangle$ *iff* $\langle v'(x) \rangle \leq \langle v'(y) \rangle$ *for all* $x, y \in C$*.*

Proposition 2 ([4]). *For a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$, the relation $\cong$ defined on states of $[\![A]\!]$ by $(\ell, v) \cong (\ell', v')$ if $\ell = \ell'$ and $v \cong v'$ is an F-sensitive time-abstracted bisimulation. The quotient $[\![A]\!]_{\cong}$ is finite.*

The equivalence classes of valuations of A with respect to $\cong$ are called *regions*, and the quotient $[\![A]\!]_{\cong}$ is called the *region automaton* associated with A.

Proposition 3 ([8]). *The number of regions for a timed automaton A with a set C of n clocks is bounded above by*

$$n! \cdot 2^n \cdot \prod_{x \in C} (2c_A(x) + 2).$$

Example 2 (continued). The 69 regions of the timed automaton A from Figure 2 are depicted in Figure 4.

Propositions 1 and 2 together now give the decidability part of the theorem below; for PSPACE-completeness see [6, 37].

Theorem 3. *The reachability problem for timed automata is* PSPACE*-complete.*

2.4 Behavioural Refinement Relations

We have already introduced time-abstracted simulations and bisimulations in Definition 8. As a corollary of Proposition 2, these are decidable:

Theorem 4. *Time-abstracted simulation and bisimulation are decidable for timed automata.*

Proof. One only needs to see that time-abstracted (bi)simulation in the timed automaton is the same as ordinary (bi)simulation in the associated region automaton; indeed, any state in $[\![A]\!]$ is untimed bisimilar to its image in $[\![A]\!]_{\cong}$. The result follows by finiteness of the region automaton. □

The following provides a *time-sensitive* variant of (bi)simulation.

Definition 12. *Let $A = (L, \ell_0, F, C, \Sigma, I, E)$ be a timed automaton. A reflexive, transitive relation $R \subseteq L \times \mathbb{R}_{\geq 0}^C \times L \times \mathbb{R}_{\geq 0}^C$ is a* timed simulation *provided that for all $(\ell_1, v_1)\ R\ (\ell_2, v_2)$,*

- *for all $(\ell_1, v_1) \xrightarrow{d} (\ell_1', v_1')$, $d \in \mathbb{R}_{\geq 0}$, there exists some (ℓ_2', v_2') such that $(\ell_1', v_1')\ R\ (\ell_2', v_2')$ and $(\ell_2, v_2) \xrightarrow{d} (\ell_2', v_2')$, and*
- *for all $(\ell_1, v_1) \xrightarrow{a} (\ell_1', v_1')$, $a \in \Sigma$, there exists some (ℓ_2', v_2') such that $(\ell_1', v_1')\ R\ (\ell_2', v_2')$ and $(\ell_2, v_2) \xrightarrow{a} (\ell_2', v_2')$.*

R is called a timed bisimulation *if it is also symmetric. Two states (ℓ_1, v_1), $(\ell_2, v_2) \in [\![A]\!]$ are said to be* timed bisimilar*, written $(\ell_1, v_1) \sim (\ell_2, v_2)$, if there exists a timed bisimulation R for which $(\ell_1, v_1)\ R\ (\ell_2, v_2)$.*

Note that, except for our requirement of reflexivity and transitivity, a timed (bi)simulation on A is the same as a standard (bi)simulation on $[\![A]\!]$.

Definition 13. *Two timed automata $A = (L^A, \ell_0^A, F^A, C^A, \Sigma^A, I^A, E^A)$ and $B = (L^B, \ell_0^B, F^B, C^B, \Sigma^B, I^B, E^B)$ are said to be* timed bisimilar*, denoted $A \sim B$, if $(\ell_0^A, v_0) \sim (\ell_0^B, v_0)$ in the disjoint-union transition system $[\![A]\!] \sqcup [\![B]\!]$.*

Timed simulation of timed automata can be analogously defined. The following decidability result was established for *parallel timed processes* in [36]; below we give a version of the proof which has been adapted for timed automata.

Theorem 5. *Timed similarity and bisimilarity are decidable for timed automata.*

Before the proof, we need a few auxiliary definitions and lemmas. The first is a product of timed transition systems which synchronizes on time, but not on actions:

Definition 14. *The* independent product *of the timed transition systems $[\![A]\!] = (S^A, s_0^A, \Sigma^A \cup \mathbb{R}_{\geq 0}, T^A)$, $[\![B]\!] = (S^B, s_0^B, \Sigma^B \cup \mathbb{R}_{\geq 0}, T^B)$ associated with timed automata A, B is $[\![A]\!] \times [\![B]\!] = (S, s_0, \Sigma^A \cup \Sigma^B \cup \mathbb{R}_{\geq 0}, T)$ given by*

$$\begin{aligned} S &= S^A \times S^B \qquad\qquad s_0 = (s_0^A, s_0^B) \\ T &= \big\{(p,q) \xrightarrow{a} (p',q) \bigm| a \in \Sigma, p \xrightarrow{a} p' \in T^A\big\} \\ &\quad \cup \big\{(p,q) \xrightarrow{b} (p,q') \bigm| b \in \Sigma, q \xrightarrow{b} q' \in T^B\big\} \\ &\quad \cup \big\{(p,q) \xrightarrow{d} (p',q') \bigm| d \in \mathbb{R}_{\geq 0}, p \xrightarrow{d} p' \in T^A, q \xrightarrow{d} q' \in T^B\big\} \end{aligned}$$

We need to extend region equivalence $\cong$ to the independent product. Below, $\oplus$ denotes vector concatenation (direct sum); note that $(p_1, q_1) \cong (p_2, q_2)$ is not the same as $p_1 \cong q_1$ and $p_2 \cong q_2$, as fractional orderings $\langle x^A \rangle \bowtie \langle x^B \rangle$, for $x^A \in C^A$, $x^B \in C^B$, have to be accounted for in the former, but not in the latter. Hence $(p_1, q_1) \cong (p_2, q_2)$ implies $p_1 \cong q_1$ and $p_2 \cong q_2$, but not vice-versa.

Definition 15. *For states $p_i = (\ell^{p_i}, v^{p_i})$ in $\llbracket A \rrbracket$ and $q_i = (\ell^{q_i}, v^{q_i})$ in $\llbracket B \rrbracket$ for $i = 1, 2$, we say that $(p_1, q_1) \cong (p_2, q_2)$ iff $\ell^{p_1} = \ell^{p_2} \wedge \ell^{q_1} = \ell^{q_2}$ and $v^{p_1} \oplus v^{q_1} \cong v^{p_2} \oplus v^{q_2}$.*

Note that the number of states in $\big(\llbracket A \rrbracket \times \llbracket B \rrbracket\big)_{\cong}$ is finite, with an upper bound given by Proposition 3. Next we define transitions in $\big(\llbracket A \rrbracket \times \llbracket B \rrbracket\big)_{\cong}$:

Notation 6. *Regions in $\big(\llbracket A \rrbracket \times \llbracket B \rrbracket\big)_{\cong}$ will be denoted X, X'. The equivalence class of a pair $(p, q) \in \llbracket A \rrbracket \times \llbracket B \rrbracket$ is denoted $[p, q]$.*

Definition 16. *For $X, X' \in \big(\llbracket A \rrbracket \times \llbracket B \rrbracket\big)_{\cong}$ we say that*

- *$X \xrightarrow{a}_{\ell} X'$ for $a \in \Sigma$ if for all $(p, q) \in X$ there exists $(p', q) \in X'$ such that $(p, q) \xrightarrow{a} (p', q)$ in $\llbracket A \rrbracket \times \llbracket B \rrbracket$,*
- *$X \xrightarrow{b}_{r} X'$ for $b \in \Sigma$ if for all $(p, q) \in X$ there exists $(p, q') \in X'$ such that $(p, q) \xrightarrow{b} (p, q')$ in $\llbracket A \rrbracket \times \llbracket B \rrbracket$, and*
- *$X \xrightarrow{\delta} X'$ if for all $(p, q) \in X$ there exists $d \in \mathbb{R}_{\geq 0}$ and $(p', q') \in X'$ such that $(p, q) \xrightarrow{d} (p', q')$.*

Definition 17. *A subset $\mathcal{B} \subseteq \big(\llbracket A \rrbracket \times \llbracket B \rrbracket\big)_{\cong}$ is a* symbolic bisimulation *provided that for all $X \in \mathcal{B}$,*

- *whenever $X \xrightarrow{a}_{\ell} X'$ for some $X' \in \big(\llbracket A \rrbracket \times \llbracket B \rrbracket\big)_{\cong}$, then $X' \xrightarrow{a}_{r} X''$ for some $X'' \in \mathcal{B}$,*
- *whenever $X \xrightarrow{a}_{r} X'$ for some $X' \in \big(\llbracket A \rrbracket \times \llbracket B \rrbracket\big)_{\cong}$, then $X' \xrightarrow{a}_{\ell} X''$ for some $X'' \in \mathcal{B}$, and*
- *whenever $X \xrightarrow{\delta} X'$ for some $X' \in \big(\llbracket A \rrbracket \times \llbracket B \rrbracket\big)_{\cong}$, then $X' \in \mathcal{B}$.*

Note that it is decidable whether $\big(\llbracket A \rrbracket \times \llbracket B \rrbracket\big)_{\cong}$ admits a symbolic bisimulation. The following proposition finishes the proof of Theorem 5.

Proposition 4. *The quotient $\big(\llbracket A \rrbracket \times \llbracket B \rrbracket\big)_{\cong}$ admits a symbolic bisimulation if and only if $A \sim B$.*

Proof (cf. [36]). For a given symbolic bisimulation $\mathcal{B} \subseteq \big(\llbracket A \rrbracket \times \llbracket B \rrbracket\big)_{\cong}$, the set $R_{\mathcal{B}} = \big\{(p, q) \bigm| [p, q] \in \mathcal{B}\big\} \subseteq \llbracket A \rrbracket \times \llbracket B \rrbracket$ is a timed bisimulation. For the other direction, one can construct a symbolic bisimulation from a timed bisimulation $R \subseteq \llbracket A \rrbracket \times \llbracket B \rrbracket$ by $\mathcal{B}_R = \big\{[p, q] \bigm| (p, q) \in R\big\}$

2.5 Language Inclusion and Equivalence

Similarly to the untimed setting, there is also a notion of language inclusion and equivalence for timed automata. We need to introduce the notion of *timed trace* first. Note that we restrict to *finite* timed traces here; similar results are available for infinite traces in timed automata with Büchi or Muller acceptance conditions, see [8].

Definition 18. *A* timed trace *over a finite set of actions Σ is a finite sequence $((t_1, a_1), (t_2, a_2), \dots, (t_k, a_k))$, where $a_i \in \Sigma$ and $t_i \in \mathbb{R}_{\geq 0}$ for $i = 1, \dots, k$, and $t_i < t_{i+1}$ for $i = 1, \dots, k-1$. The set of all timed traces over Σ is denoted $T\Sigma^*$.*

In a pair (t_i, a_i), the number t_i is called the *time stamp* of the action a_i, *i.e.* the time at which event a_i occurs.

Remark 3. Timed traces as defined above are also known as *strongly monotonic* timed traces, because of the assumption that no consecutive events occur at the same time. *Weakly* monotonic timed traces, *i.e.* with requirement $t_i \leq t_{i+1}$ instead of $t_i < t_{i+1}$, have also been considered, and there are some subtle differences between the two; see [65] for an important example.

Definition 19. *A timed trace $((t_1, a_1), \dots, (t_k, a_k))$ is* accepted *by a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$ if there is an accepting run*

$$(\ell_0, v_0) \xrightarrow{t_1} (\ell_0, v_0 + t_1) \xrightarrow{a_1} (\ell_1, v_1) \xrightarrow{t_2 - t_1} \cdots$$
$$\cdots \xrightarrow{a_{k-1}} (\ell_{k-1}, v_{k-1}) \xrightarrow{t_k - t_{k-1}} (\ell_{k-1}, v_{k-1} + t_k - t_{k-1}) \xrightarrow{a_k} (\ell_k, v_k)$$

in A. The timed language *of A is $L(A) = \{\tau \in T\Sigma^* \mid \tau$ accepted by $A\}$.*

It is clear that $L(A) = \emptyset$ if and only if none of the locations in F is reachable, hence Theorem 3 provides us with the decidability result in the following theorem. Undecidability of universality was established in [8]; we give an account of the proof below.

Theorem 7. *For a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$, deciding whether $L(A) = \emptyset$ is* PSPACE*-complete. It is undecidable whether $L(A) = T\Sigma^*$.*

Proof. We may show that the universality problem for a timed automata is undecidable by reduction from the Σ_1^1-hard problem of deciding whether a given 2-counter machine M has a recurring computation.

Let the timed language L_u be the set of timed traces encoding recurring computations of M. Observe that $L_u = \emptyset$ if and only if M does not have such a computation. We then construct a timed automaton A_u which accepts the complement of L_u, *i.e.* $L(A_u) = T\Sigma^* \setminus L_u$. Hence the language of A_u is universal if and only if M does not have a recurring computation.

Recall that a 2-counter, or Minsky, machine M is a finite sequence of labeled instructions $\{I_0, \cdots, I_n\}$ and counters $\mathtt{x}_1$ and $\mathtt{x}_2$, with I_i for $0 \leq i \leq n-1$ on the form

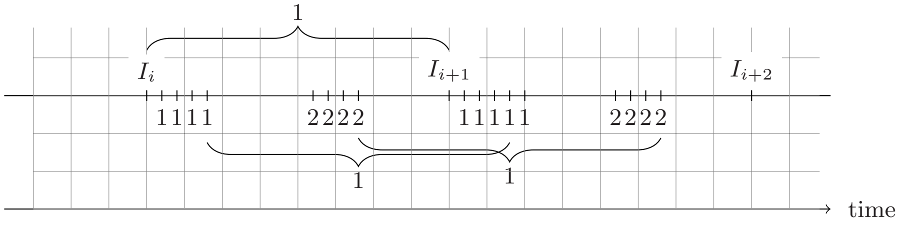


Fig. 5. Timed trace encoding a increment instruction I_{i+1} of a 2-counter machine

$$I_i : \mathtt{x}_c \texttt{ := } \mathtt{x}_c + \mathtt{1}\texttt{; goto } I_j \quad \mathit{or} \quad I_i : \begin{cases} \texttt{if } \mathtt{x}_c \texttt{ = 0 then goto } I_j \\ \texttt{else } \mathtt{x}_c \texttt{ = } \mathtt{x}_c\texttt{-1; goto } I_k \end{cases}$$

for $c \in 1, 2$, with a special I_n : `Halt` instruction which stops the computation.

The language L_u is designed such that each I_i and the counters $\mathtt{x}_1$ and $\mathtt{x}_2$ are represented by actions in Σ. A correctly encoded computation is represented by a timed trace where "instruction actions" occur at discrete intervals, while the state (values of $\mathtt{x}_1$ and $\mathtt{x}_2$) is encoded by occurrences of "counter actions" in-between instruction actions (*e.g.* if $\mathtt{x}_i = 5$ after instruction I_j, then action x_i occurs 5 times within the succeeding interval of length 1).

When counters are incremented (or decremented), one more (or less) such action occurs through the next interval, and increments and decrements are always from the right. Additionally we require corresponding counter actions to occur exactly with a time difference of 1, such that if x_i occurs with time stamp a then also x_i occurs with time stamp $a + 1$, unless x_i is the rightmost x_i action and I_i at time stamp $\lfloor a \rfloor$ is a decrement of $\mathtt{x}_i$. Figure 5 shows a increment of $\mathtt{x}_1$ (from 4 to 5) using actions 1 and 2.

We obtain A_u as a disjunction of timed automata $A^1, \ldots, A^k$ where each A^i violates some property of a (correctly encoded) timed trace in L_u, either by accepting traces of incorrect format or inaccurate encodings of instructions.

Consider the instruction: `(p):` $\mathtt{x}_1$`:=` $\mathtt{x}_1$`+1 goto (q)`, incrementing $\mathtt{x}_1$ and jumping to `q`. A correct encoding would be similar to the one depicted in Figure 5 where all 1's and 2's are matched *one* time unit later, but with an additional 1 action occurring. In order to accept all traces except this encoding we must consider all possible violations, *i.e.*

- not incrementing the counter (no change),
- decrementing the counter,
- incrementing the counter more than once,
- jumping to the wrong instruction, or
- incrementing the wrong counter,

and construct a timed automaton having exactly such traces.

Figure 6 shows the timed automaton accepting traces in which instruction `p` yields no change of $\mathtt{x}_1$. □

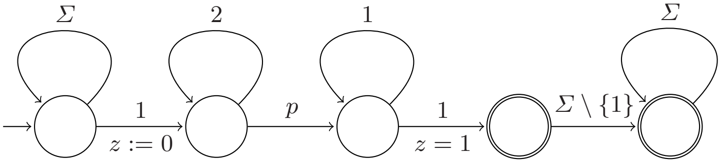


Fig. 6. Timed automaton which violates the encoding of the increment instruction

Turning our attention to timed trace inclusion and equivalence, we note the following.

Proposition 5. *Let A and B be timed automata. If A is timed simulated by B, then $L(A) \subseteq L(B)$. If A and B are timed bisimilar, then $L(A) = L(B)$.*

By a standard argument, Theorem 7 implies undecidability of timed trace inclusion and equivalence, a result first shown in [7].

Theorem 8. *Timed trace inclusion and equivalence are undecidable for timed automata.*

There is also a notion of *untimed* traces for timed automata.

Definition 20. *The* untiming *of a set of timed traces $L \subseteq T\Sigma^*$ over a finite set of actions Σ is the set*

$$UL = \big\{ w = (a_1, \dots, a_k) \in \Sigma^* \; \big| \; \exists t_1, \dots, t_k \in \mathbb{R}_{\geq 0} : ((t_1, a_1), \dots, (t_k, a_k)) \in L \big\}.$$

Hence we have a notion of the set $UL(A)$ of *untimed language* of a timed automaton A. One can also define an untime operation U for timed automata, forgetting about the timing information of a timed automaton and thus converting it to a finite automaton; note however that $UL(A) \subsetneq L(UA)$ in general.

Lemma 1 ([8]). *For A a timed automaton, $UL(A) = L(\llbracket A \rrbracket_{\cong})$ provided that δ-transitions in $\llbracket A \rrbracket_{\cong}$ are taken as silent.*

As a corollary, sets of untimed traces accepted by timed automata are *regular*:

Theorem 9 ([8]). *For a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$, the set $UL(A) \subseteq \Sigma^*$ is regular. Accordingly, whether $UL(A) = \emptyset$ is decidable, and so is whether $UL(A) = \Sigma^*$. Also untimed trace inclusion and equivalence are decidable.*

2.6 Zones and Difference-Bound Matrices

As shown in the above sections, regions provide a finite and elegant abstraction of the infinite state space of timed automata, enabling us to prove decidability of reachability, timed and untimed bisimilarity, untimed language equivalence and language emptiness.

$$Z = \begin{cases} x_1 \le 3 \\ x_1 - x_2 \le 10 \\ x_1 - x_2 \ge 4 \\ x_1 - x_3 \le 2 \\ x_3 - x_2 \le 2 \\ x_3 \ge -5 \end{cases}$$

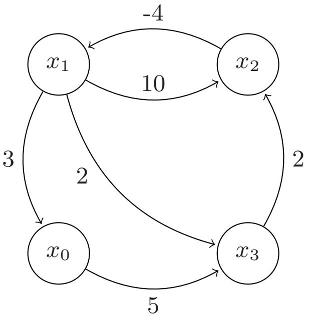


Fig. 7. Graph representation of extended clock constraint

Unfortunately, the number of states obtained from the region partitioning is extremely large. In particular, by Proposition 3 the number of regions is exponential in the number of clocks as well as in the maximal constants of the timed automaton. Efforts have been made in developing more efficient representations of the state space [20, 25, 56], using the notion of *zones* from Definition 4 as a coarser and more compact representation of the state space.

An extended clock constraint over a finite set C may be represented using a directed weighted graph, where the nodes correspond to the elements of C together with an extra "zero" node x_0, and an edge $x_i \xrightarrow{k} x_j$ corresponds to a constraint $x_i - x_j \le k$ (if there is more than one upper bound on $x_i - x_j$, k is the minimum of all these constraints' right-hand sides). The extra clock x_0 is fixed at value 0, so that a constraint $x_i \le k$ can be represented as $x_i - x_0 \le k$. Lower bounds on $x_i - x_j$ are represented as (possibly negative) upper bounds on $x_j - x_i$, and strict bounds $x_i - x_j < k$ are represented by adding a flag to the corresponding edge.

The weighted graph in turn may be represented by its adjacency matrix, which is known as a *difference-bound matrix* or DBM. The above technique has been introduced in [39].

Example 3. Figure 7 gives an illustration of an extended clock constraint together with its representation as a difference-bound matrix. Note that the clock constraint contains superfluous information.

Zone-based reachability analysis of a timed automaton A uses symbolic states of the type (ℓ, Z), where ℓ is a location of A and Z is a zone, instead of the region-based symbolic states of Proposition 2.

Definition 21. *For a finite set C, $Z \subseteq \mathbb{R}_{\ge 0}^C$, and $r \subseteq C$, define*

- *the* delay *of Z by $Z^\uparrow = \{v + d \mid v \in Z, d \in \mathbb{R}_{\ge 0}\}$ and*
- *the* reset *of Z under r by $Z[r] = \{v[r] \mid v \in Z\}$.*

Lemma 2 ([46, 69]). *If Z is a zone over C and $r \subseteq C$, then $Z^\uparrow$ and $Z[r]$ are also zones over C.*

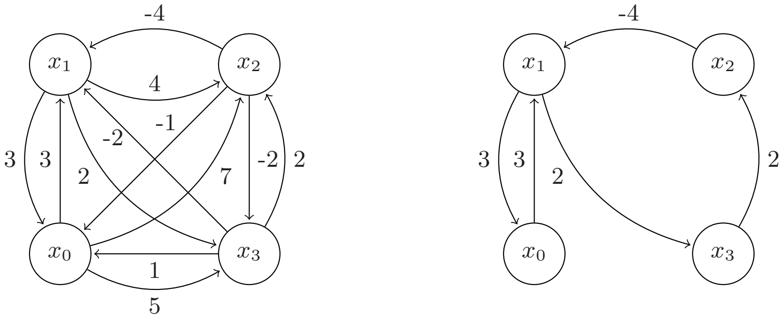


Fig. 8. Canonical representations. Left: shortest-path closure; right: shortest-path reduction.

Extended clock constraints representing $Z^{\uparrow}$ and $Z[r]$ may be computed efficiently (*i.e.* in time cubic in the number of clocks in C) by representing the zone Z in a canonical form obtained by computing the *shortest-path closure* of the directed graph representation of Z, see [54].

Example 3 (continued). Figure 8 shows two canonical representations of the difference-bound matrix for the zone Z of Figure 7. The left part illustrates the shortest-path closure of Z; on the right is the *shortest-path reduction* [54] of Z, essentially obtained by removing redundant edges from the shortest-path closure. The latter is useful for checking zone inclusion, see below.

The *zone automaton* associated with a timed automaton is similar to the region automaton of Proposition 2, but uses zones for symbolic states instead of regions:

Definition 22. *The* zone automaton *associated with a timed automaton* $A = (L, \ell_0, F, C, \Sigma, I, E)$ *is the transition system* $[\![A]\!]_Z = (S, s_0, \Sigma \cup \{\delta\}, T)$ *given as follows:*

$$S = \big\{(\ell, Z) \bigm| \ell \in L, Z \subseteq \mathbb{R}_{\geq 0}^C \textit{ zone}\big\} \qquad\qquad s_0 = \big(\ell_0, [\![v_0]\!]\big)$$

$$T = \big\{(\ell, Z) \overset{\delta}{\rightsquigarrow} \big(\ell, Z^{\uparrow} \wedge I(\ell)\big)\big\}$$

$$\cup \big\{(\ell, Z) \overset{a}{\rightsquigarrow} \big(\ell', (Z \wedge \varphi)[r] \wedge I(\ell')\big) \bigm| \ell \xrightarrow{\varphi, a, r} \ell' \in E\big\}$$

The analogue of Proposition 1 for zone automata is as follows:

Proposition 6 ([69]). *A state (ℓ, v) in a timed automaton $A = (L, \ell_0, F, C, \Sigma, I, E)$ is reachable if and only if there is a zone $Z \subseteq \mathbb{R}_{\geq 0}^C$ for which $v \in Z$ and such that (ℓ, Z) is reachable in $[\![A]\!]_Z$.*

The zone automaton associated with a given timed automaton is *infinite* and hence unsuitable for reachability analysis. Finiteness can be enforced by employing *normalization*, using the fact that region equivalence $\cong$ has finitely many equivalence classes:

Definition 23. *For a timed automaton A and a zone $Z \subseteq \mathbb{R}_{\geq 0}^{C}$, the* normalization *of Z is the set $\{v : C \to \mathbb{R}_{\geq 0} \mid \exists v' \in D : v \cong v'\}$*

The normalized zone automaton is defined in analogy to the zone automaton from above, and the analogue of Proposition 6 holds for the normalized zone automaton. Hence we can obtain a reachability algorithm by applying any search strategy (depth-first, breadth-first, or another) on the normalized zone automaton.

Remark 4. For timed automata on *extended* clock constraints, *i.e.* with diagonal constraints permitted, it can be shown [24, 27] that normalization as defined above does *not* give rise to a sound and complete characterization of reachability. Instead, one can apply a refined normalization which depends on the difference constraints used in the timed automaton, see [24].

In addition to the efficient computation of symbolic successor states according to the $\rightsquigarrow$ relation, termination of reachability analysis requires that we can efficiently recognize whether the search algorithm has encountered a given symbolic state. Here it is crucial that there is an efficient way of deciding inclusion $Z_1 \subseteq Z_2$ between zones. Both the shortest-path-closure canonical form as well as the more space-economical shortest-path-reduced canonical form [54], *cf.* Example 3, allow for efficient inclusion checking.

In analogy to difference-bound matrices and overcoming some of their problems, the data structure called *clock difference diagram* has been proposed [56]. However, the design of efficient algorithms for delay and reset operations over that data structure is a challenging open problem; generally, the design of efficient data structures for computations with (unions of) zones is a field of active research, see [3, 11, 64] for some examples.

3 Weighted Timed Automata

The notion of *weighted* — or *priced* — timed automata was introduced independently, at the very same conference, by Behrmann *et.al.* [18] and Alur *et.al.* [9]. In these models both edges and locations can be decorated with weights, or prices, giving the cost of taking an action transition or the cost per time unit of delaying in a given location. The total cost of a trace is then simply the accumulated (or total) weight of its discrete and delay transitions.

As a first result, the above two papers independently, and with quite different methods, showed that the problem of cost-optimal reachability is computable for weighted timed automata with non-negative weights. Later, optimal reachability for timed automata with several weight functions was considered in [59] as well as optimal infinite runs in [29, 41].

Definition 24. *A* weighted timed automaton *is a tuple $A = (L, \ell_0, F, C, \Sigma, I, E, R, P)$, where $(L, \ell_0, F, C, \Sigma, I, E)$ is a timed automaton, $R : L \to \mathbb{Z}$ a location weight-rate mapping, and $P : E \to \mathbb{Z}$ an edge weight mapping.*

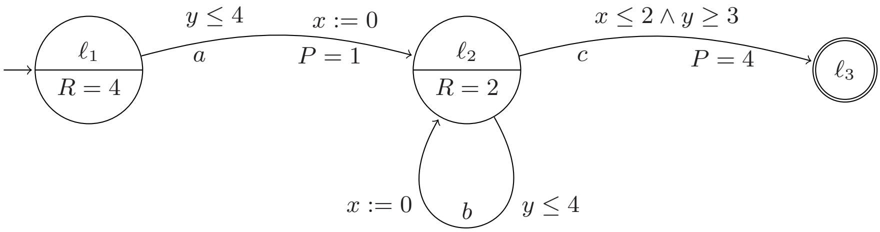


Fig. 9. A weighted timed automaton with two clocks

The semantics *of A is the weighted transition system $[\![A]\!] = (S, s_0, \Sigma \cup \mathbb{R}_{\geq 0}, T, w)$, where $(S, s_0, \Sigma \cup \mathbb{R}_{\geq 0}, T)$ is the semantics of the underlying timed automaton $(L, \ell_0, F, C, \Sigma, I, E)$, and the transition weights $w : T \to \mathbb{R}$ are given as follows:*

$$w\big((\ell, v) \xrightarrow{d} (\ell, v+d)\big) = d\,R(\ell)$$
$$w\big((\ell, v) \xrightarrow{a} (\ell', v')\big) = P\big(\ell \xrightarrow{\varphi, a, r} \ell'\big) \quad \text{with } v \models \varphi, v' = v[r]$$

We shall denote weighted edges and transitions by symbols $\xrightarrow[w]{e}$ to illustrate an edge or a transition labeled e with weight w.

3.1 Optimal Reachability

The objective of optimal reachability analysis is to find runs to a final location with the lowest *total weight* as defined below.

Example 4. Figure 9 shows a simple weighted timed automaton with final location ℓ_3. Below we give a few examples of accepting runs, where we identify valuations $v : \{x, y\} \to \mathbb{R}_{\geq 0}$ with their values $(v(x), v(y))$. The total weights of the runs given here are 17 and 11; actually the second run is *optimal* in the sense of Problem 2 below:

$$(\ell_1, 0, 0) \xrightarrow[12]{3} (\ell_1, 3, 3) \xrightarrow[1]{a} (\ell_2, 0, 3) \xrightarrow[4]{c} (\ell_3, 0, 3)$$
$$(\ell_1, 0, 0) \xrightarrow[1]{a} (\ell_2, 0, 0) \xrightarrow[6]{3} (\ell_2, 3, 3) \xrightarrow[0]{b} (\ell_2, 0, 3) \xrightarrow[4]{c} (\ell_3, 0, 3)$$

Definition 25. *The* total weight *of a finite run $\rho = s_0 \xrightarrow[w_1]{} s_1 \xrightarrow[w_2]{} \cdots \xrightarrow[w_k]{} s_k$ in a weighted transition system is $w(\rho) = \sum_{i=1}^k w_k$.*

We are now in a position to state the problem with which we are concerned here: We want to find accepting runs with minimum total weight in a weighted timed automaton A. However due to the possible use of strict clock constraints on edges and in locations of A, the minimum total weight might not be realizable, *i.e.* there might be no run which achieves it. For this reason, one also needs to consider (infinite) *sets* of runs and the infimum of their members' total weights:

Problem 2 (Optimal reachability). Given a weighted timed automaton A, compute $W = \inf \{w(\rho) \mid \rho \text{ accepting run in } A\}$ and a set P of accepting runs for which $\inf_{\rho \in P} w(\rho) = W$.

The key ingredient in the proof of the following theorem is the introduction of *weighted regions* in [18]. A weighted region is a region as of Definition 11 enriched with an affine cost function describing in a finite manner the cost of reaching any point within it. This notion allows one to define the weighted region automaton associated with a weighted timed automaton, and one can then show that optimal reachability can be computed in the weighted region automaton. PSPACE-hardness in the below theorem follows from PSPACE-hardness of reachability for timed automata.

Theorem 10 ([18]). *The optimal reachability problem for weighted timed automata with non-negative weights is* PSPACE*-complete.*

Similar to the notion of regions for timed automata, the number of weighted regions is exponential in the number of clocks as well as in the maximal constants of the timed automaton. Hence a notion of *weighted zone* — a zone extended with an affine cost function — was introduced [53] together with an efficient, symbolic A^*-algorithm for searching for cost-optimal tracing using branch-and-bound techniques. In particular, efficient means of generalizing the notion of symbolic successor to incorporate the affine cost functions were given.

During the symbolic exploration, several small linear-programming problems in terms of determining the minimal value of the cost function over the given zone have to be dealt with. Given that the constraints of these problems are simple difference constraints, it turns out that substantial gain in performance may be achieved by solving the dual problem of minimum-cost flow [67]. The newly emerged branch UppAal Cora provides an efficient tool for cost-optimal reachability analysis, applying the above data structures and algorithms and allowing the user to guide and heuristically prune the search.

3.2 Multi-weighted Timed Automata

The below formalism of doubly weighted timed automata is a generalization of weighted timed automata useful for modeling systems with several different resources.

Definition 26. *A* doubly weighted timed automaton *is a tuple*

$$A = (L, \ell_0, F, C, \Sigma, I, E, R, P)$$

where $(L, \ell_0, F, C, \Sigma, I, E)$ is a timed automaton, $R : L \to \mathbb{Z}^2$ a location weight-rate mapping, and $P : E \to \mathbb{Z}^2$ an edge weight mapping.

The semantics of a doubly weighted timed automaton is a doubly weighted transition system defined similarly to Definition 24, and the total weight of finite

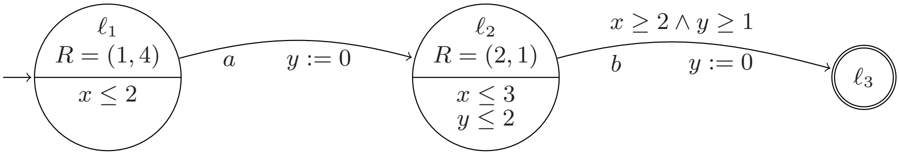


Fig. 10. A doubly weighted timed automaton with two clocks

runs is defined accordingly as a pair; we shall refer to the total weights as w_1 and w_2 respectively. These definitions have natural generalizations to *multi-weighted* timed automata with more than two weight coordinates.

The objective of conditional reachability analysis is to find runs to a final location with the lowest total weight in the first weight coordinate while satisfying a constraint on the other weight coordinate.

Example 5. Figure 10 depicts a simple doubly weighted timed automaton with final location ℓ_3. Under the constraint $w_2 \le 3$, the optimal run of the automaton can be seen to be

$$(\ell_1, 0, 0) \xrightarrow[(\frac{1}{3}, \frac{4}{3})]{1/3} (\ell_1, 1/3, 1/3) \xrightarrow{a} (\ell_2, 1/3, 0) \xrightarrow[(\frac{10}{3}, \frac{5}{3})]{5/3} (\ell_2, 2, 5/3) \xrightarrow{b} (\ell_3, 2, 0)$$

with total weight $\big(\frac{11}{3}, 3\big)$.

The precise formulation of the conditional optimal reachability problem is as follows, where we again need to refer to (possibly infinite) sets of runs:

Problem 3 (Conditional optimal reachability). Given a doubly weighted timed automaton A and $M \in \mathbb{Z}$, compute $W = \inf \big\{ w_1(\rho) \bigm| \rho \text{ accepting run in } A, w_2(\rho) \le M \big\}$ and a set P of accepting runs such that $w_2(\rho) \le M$ for all $\rho \in P$ and $\inf_{\rho \in P} w(\rho) = W$.

Theorem 11 ([58,59]). *The conditional optimal reachability problem is computable for doubly weighted timed automata with non-negative weights and without weights on edges.*

The proof of the above theorem rests on a direct generalization of weighted to *doubly-weighted* zones. An extension can be found in [59], where it is shown that also the *Pareto frontier*, *i.e.* the set of cost vectors which cannot be improved in any cost variable, can be computed.

3.3 Optimal Infinite Runs

In this section we shall be concerned with computing optimal *infinite* runs in (doubly) weighted timed automata. We shall treat both the *limit ratio* viewpoint discussed in [29] and the *discounting* approach of [41].

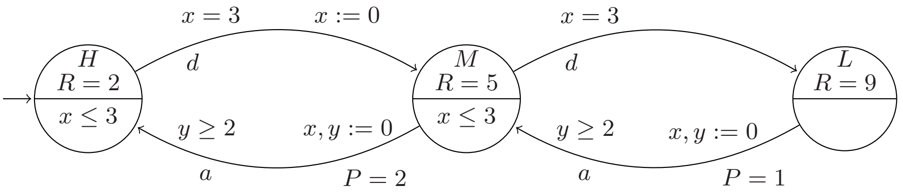


Fig. 11. A weighted timed automaton modelling a simple production system

Example 6. Figure 11 shows a simple production system modelled as a weighted timed automaton. The system has three modes of production, High, Medium, and Low. The weights model the *cost* of production, so that the High production mode has a low cost, which is preferable to the high cost of the Low production mode. After operating in a High or Medium production mode for three time units, production automatically degrades (action d) to a lower mode. When in Medium or Low production mode, the system can be attended to (action a), which advances it to a higher mode.

The objective of *optimal-ratio analysis* is to find an infinite run in a doubly weighted timed automaton which minimizes the *ratio* between the two total weights. This will be formalized below.

Definition 27. *The* total ratio *of a finite run* $\rho = s_0 \xrightarrow[z_1]{w_1} s_1 \xrightarrow[z_2]{w_2} \cdots \xrightarrow[z_k]{w_k} s_k$ *in a doubly weighted transition system is*

$$\Gamma(\rho) = \frac{\sum_{i=1}^{k} w_k}{\sum_{i=1}^{k} z_k}.$$

The total ratio of an infinite run $\rho = s_0 \xrightarrow[z_1]{w_1} s_1 \xrightarrow[z_2]{w_2} \cdots$ *is*

$$\Gamma(\rho) = \liminf_{k \to \infty} \Gamma(s_0 \to \cdots \to s_k).$$

A special case of optimal-ratio analysis is given by weight-per-time models, where the interest is in minimizing total weight per accumulated time. The example provided in this section is a case of this. In the setting of optimal-ratio analysis, these can be modelled as doubly weighted timed automata with $R_2(\ell) = 1$ and $P_2(e) = 0$ for all locations ℓ and edges e.

Example 6 (continued). In the timed automaton of Figure 11, the following cyclic behaviour provides an infinite run ρ:

$$(H, 0, 0) \xrightarrow{3} (H, 3, 3) \xrightarrow{d} (M, 0, 3) \xrightarrow{3} (M, 3, 6) \xrightarrow{d} (L, 3, 6) \xrightarrow{1}$$
$$(L, 4, 7) \xrightarrow{a} (M, 0, 0) \xrightarrow{3} (M, 3, 3) \xrightarrow{a} (H, 0, 0) \to \cdots$$

Taking the weight-per-time viewpoint, the total ratio of ρ is $\Gamma(\rho) = 4.8$.

Problem 4 (Minimum infinite ratio). Given a doubly weighted timed automaton A, compute $W = \inf \big\{ \varGamma(\rho) \bigm| \rho \text{ infinite run in } A \big\}$ and a set P of infinite runs for which $\inf_{\rho \in P} \varGamma(\rho) = W$.

The main tool in the proof of the following theorem is the introduction of the *corner-point abstraction* of a timed automaton in [29]. This is a finite refinement of the region automaton of Definition 11 in which one also keeps track of the corner points of regions. One can then show that any infinite run with minimum ratio must pass through corner points of regions, hence these can be found in the corner-point abstraction by an algorithm first proposed in [51].

The technical condition in the theorem that the second weight coordinate be *strongly diverging* means that any infinite run ρ in the closure of the timed automaton in question satisfies $w_2(\rho) = \infty$, see [29] for details.

Theorem 12 ([29]). *The minimum infinite ratio problem is computable for doubly weighted timed automata with non-negative and strongly diverging second weight coordinate.*

For *discount-optimal analysis*, the objective is to find an infinite run in a weighted timed automaton which minimizes the *discounted total weight* as defined below. The point of discounting is that the weight of actions is discounted with time, so that the impact of an event decreases, the further in the future it takes place.

In the definition below, ε is the empty run, and $(\ell, v) \to \rho$ denotes the concatenation of the transition $(\ell, v) \to$ with the run ρ.

Definition 28. *The* discounted total weight *of finite runs in a weighted timed automaton under discounting factor $\lambda \in [0, 1[$ is given inductively as follows:*

$$\begin{aligned} w_\lambda(\varepsilon) &= 0 \\ w_\lambda\big((\ell, v) \xrightarrow[P]{a} \rho\big) &= P + w_\lambda(\rho) \\ w_\lambda\big((\ell, v) \xrightarrow{d} \rho\big) &= R(\ell) \int_0^d \lambda^\tau \, d\tau + \lambda^d w_\lambda(\rho) \end{aligned}$$

The discounted total weight of an infinite run $\rho = (\ell_0, v_0) \xrightarrow{d_1} (\ell_0, v_0 + d_1) \xrightarrow[P_1]{a_1} (\ell_1, v_1) \to \cdots$ is

$$w_\lambda(\rho) = \lim_{k \to \infty} w_\lambda\big((\ell_0, v_0) \to \cdots \xrightarrow[P_k]{a_k} (\ell_k, v_k)\big)$$

provided that the limit exists.

Example 6 (continued). The discounted total weight of the infinite run ρ in the timed automaton of Figure 11 satisfies the following equality, where $I_t = \int_0^t \lambda^\tau \, d\tau = -\frac{1}{\ln \lambda}(1 - \lambda^t)$:

$$w_\lambda(\rho) = 2I_3 + \lambda^3(5I_3 + \lambda^3(9I_1 + \lambda(1 + 5I_3 + \lambda^3(2 + w_\lambda(\rho)))))$$

With a discounting factor of $\lambda = .9$ for example, the discounted total weight of ρ would hence be $w_\lambda(\rho) \approx 40.5$.

Problem 5 (Minimum discounted weight). Given a weighted timed automaton A and $\lambda \in [0,1[$, compute $W = \inf \{ w_\lambda(\rho) \mid \rho \text{ infinite run in } A \}$ and a set P of infinite runs for which $\inf_{\rho \in P} w_\lambda(\rho) = W$.

The proof of the following theorem rests again on the corner-point abstraction, and on a result in [10]. The technical condition that the timed automaton be time-divergent is analogous to the condition on the second weight coordinate in Theorem 12.

Theorem 13 ([41]). *The minimum discounted weight problem is computable for time-divergent weighted timed automata with non-negative weights and rational λ.*

4 Timed Games

Recently, substantial effort has been made towards the synthesis of winning strategies for timed games with respect to *safety* and *reachability control objectives.* From known region-based decidability results, efficient on-the-fly algorithms have been developed [34, 68] and implemented in the newest branch UppAal Tiga.

For timed games, as for untimed ones, transitions are either controllable or uncontrollable (*i.e.* under the control of an environment), and the problem is to synthesize a strategy for *when* to take *which* (enabled) controllable transitions in order that a given objective is guaranteed regardless of the behaviour of the environment.

Definition 29. *A* timed game *is a tuple $(L, \ell_0, F, C, \Sigma_c, \Sigma_u, I, E)$ with $\Sigma_c \cap \Sigma_u = \emptyset$ and for which the tuple $(L, \ell_0, F, C, \Sigma = \Sigma_c \cup \Sigma_u, I, E)$ is a timed automaton.*

Edges with actions in Σ_c are said to be *controllable*, those with actions in Σ_u are *uncontrollable.*

Example 7. Figure 12 provides a simple example of a timed game. Here, $\Sigma_c = \{c_1, c_2, c_4\}$ and $\Sigma_2 = \{u_1, u_2, u_3\}$, and the controllable edges are drawn with solid lines, the uncontrollable ones with dashed lines.

We only treat *reachability* games here, where the goal of the game is to reach a final location. There is also a somewhat dual notion of *safety* games, where one instead wants to *avoid* final locations, see [34] for details.

We need the notion of *strategy*; essentially, a strategy provides instructions for which controllable edge to take, or whether to wait, in a given state:

Definition 30. *A* strategy *for a timed game $A = (L, \ell_0, F, C, \Sigma_c, \Sigma_u, I, E)$ is a partial mapping σ from finite runs of A to $\Sigma_c \cup \{\delta\}$, where $\delta \notin \Sigma$, such that for any run $\rho = (\ell_0, v_0) \to \cdots \to (\ell_k, v_k)$,*

- *if $\sigma(\rho) = \delta$, then $(\ell, v) \xrightarrow{d} (\ell, v + d)$ in $[\![A]\!]$ for some $d > 0$, and*
- *if $\sigma(\rho) = a$, then $(\ell, v) \xrightarrow{a} (\ell', v')$ in $[\![A]\!]$.*

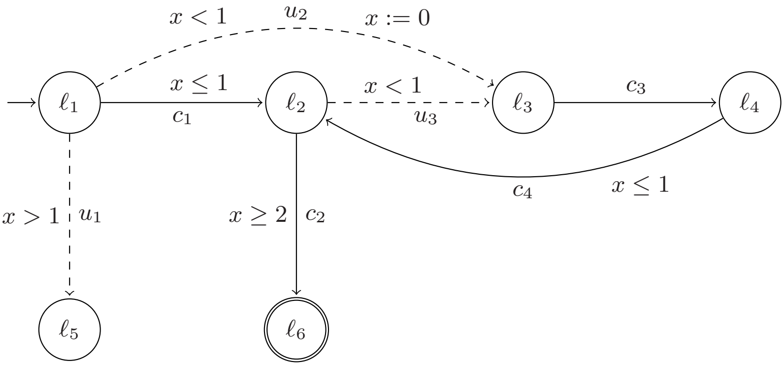


Fig. 12. A timed game with one clock. Controllable edges (with actions from Σ_c) are solid, uncontrollable edges (with actions from Σ_u) are dashed.

A strategy σ is said to be memoryless *if $\sigma(\rho)$ only depends on the last state of ρ, i.e. if $\rho_1 = (\ell_0, v_0) \xrightarrow{d_1} (\ell_0, v_0 + d_1) \to \cdots \to (\ell_k, v_k)$, $\rho_2 = (\ell_0, v_0) \xrightarrow{d_1'} (\ell_0, v_0 + d_1') \to \cdots \to (\ell_k, v_k)$ imply $\sigma(\rho_1) = \sigma(\rho_2)$.*

An *outcome* of a strategy is any run which adheres to its instructions in the obvious manner:

Definition 31. *A run $(\ell_0, v_0) \xrightarrow{d_1} (\ell_0, v_0+d_1) \to \cdots \to (\ell_k, v_k)$ in a timed game $A = (L, \ell_0, F, C, \Sigma_c, \Sigma_u, I, E)$ is said to be an* outcome *of a strategy σ provided that*

- *for all $(\ell_i, v_i) \xrightarrow{d} (\ell_i, v_i + d)$ and for all $d' < d$, we have $\sigma\big((\ell_0, v_0) \to \cdots \to (\ell_i, v_i + d')\big) = \delta$, and*
- *for all $(\ell_i, v_i + d) \xrightarrow{a} (\ell_{i+1}, v_{i+1})$ for which $a \in \Sigma_c$, we have $\sigma\big((\ell_0, v_0) \to \cdots \to (\ell_i, v_i')\big) = a$.*

An outcome is said to be maximal *if $\ell_k \in F$, or if $(\ell_k, v_k) \xrightarrow{a} (\ell_{k+1}, v_{k+1})$ implies $a \in \Sigma_u$.*

Hence an outcome is maximal if it stops in a final state, or if no controllable actions are available at its end. An underlying assumption is that uncontrollable actions cannot be forced, hence a maximal outcome which does not end in a final state may “get stuck” in a non-final state. The aim of reachability games is to find strategies all of whose maximal outcomes end in a final state:

Definition 32. *A strategy is said to be* winning *if any of its maximal outcomes is an accepting run.*

Example 7 (continued). The following memoryless strategy is winning for the reachability game on the timed game from Figure 12:

$$\sigma(\ell_1, v) = \begin{cases} \delta & \text{if } v(x) \neq 1 \\ c_1 & \text{if } v(x) = 1 \end{cases} \qquad \sigma(\ell_2, v) = \begin{cases} \delta & \text{if } v(x) < 2 \\ c_2 & \text{if } v(x) \geq 2 \end{cases}$$

$$\sigma(\ell_3, v) = \begin{cases} \delta & \text{if } v(x) < 1 \\ c_3 & \text{if } v(x) \geq 1 \end{cases} \qquad \sigma(\ell_4, v) = \begin{cases} \delta & \text{if } v(x) \neq 1 \\ c_4 & \text{if } x(x) = 1 \end{cases}$$

Problem 6 (Reachability game). Given a timed game A, does there exist a winning strategy for A?

An important ingredient in the proof of the following theorem is the fact that for reachability (as well as safety) games, it is sufficient to consider *memoryless* strategies. This is not the case for other, more subtle, control objectives (*e.g.* counting properties modulo some N) as well as for the synthesis of winning strategies under *partial observability.*

Theorem 14 ([12, 63]). *The reachability game is decidable for timed games.*

In [35] the on-the-fly algorithm applied in UppAal Tiga has been extended to timed games under partial observability.

The field of timed games is a very active research area. Research has been conducted towards the synthesis of *optimal* winning strategies for reachability games on *weighted timed games.* In [5, 30] computability of optimal strategies is shown under a certain condition of *strong cost non-zenoness*, requiring that the total weight diverges with a given minimum rate per time. Later undecidability results [28,33] show that for weighted timed games with three or more clocks this condition (or a similar one) is necessary. Lately [31] proves that optimal reachability strategies are computable for one-clock weighted timed games, though there is an unsettled (large) gap between the known lower bound complexity P and an upper bound of 3EXPTIME.

References

1. Abdeddaïm, Y., Kerbaa, A., Maler, O.: Task graph scheduling using timed automata. In: IPDPS, p. 237. IEEE Computer Society, Los Alamitos (2003)
2. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Reactive Systems: Modeling, Specification and Verification. Cambridge University Press, Cambridge (2007)
3. Allamigeon, X., Gaubert, S., Goubault, E.: Inferring min and max invariants using max-plus polyhedra. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 189–204. Springer, Heidelberg (2008)
4. Alur, R.: Timed automata. In: Halbwachs, Peled [43], pp. 8–22
5. Alur, R., Bernadsky, M., Madhusudan, P.: Optimal reachability for weighted timed games. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 122–133. Springer, Heidelberg (2004)
6. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking for real-time systems. In: LICS, pp. 414–425. IEEE Computer Society, Los Alamitos (1990)
7. Alur, R., Dill, D.L.: Automata for modeling real-time systems. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990)

8. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
9. Alur, R., Torre, S.L., Pappas, G.J.: Optimal paths in weighted timed automata. In: Benedetto, Sangiovanni-Vincentelli [23], pp. 49–62
10. Andersson, D.: Improved combinatorial algorithms for discounted payoff games. Master's thesis, Uppsala University, Department of Information Technology (2006)
11. Asarin, E., Bozga, M., Kerbrat, A., Maler, O., Pnueli, A., Rasse, A.: Data-structures for the verification of timed automata. In: Maler, O. (ed.) HART 1997. LNCS, vol. 1201, pp. 346–360. Springer, Heidelberg (1997)
12. Asarin, E., Maler, O., Pnueli, A.: Symbolic controller synthesis for discrete and timed systems. In: Antsaklis, P.J., Kohn, W., Nerode, A., Sastry, S.S. (eds.) HS 1994. LNCS, vol. 999, pp. 1–20. Springer, Heidelberg (1995)
13. Behrmann, G., Bengtsson, J., David, A., Larsen, K.G., Pettersson, P., Yi, W.: Uppaal implementation secrets. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, p. 3. Springer, Heidelberg (2002)
14. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone based abstractions of timed automata. In: Jensen, Podelski [49], pp. 312–326
15. Behrmann, G., Brinksma, E., Hendriks, M., Mader, A.: Production scheduling by reachability analysis - a case study. In: IPDPS. IEEE Computer Society, Los Alamitos (2005)
16. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: Uppaal-tiga: Time for playing games! In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 121–125. Springer, Heidelberg (2007)
17. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
18. Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J., Vaandrager, F.W.: Minimum-cost reachability for priced timed automata. In: Benedetto, Sangiovanni-Vincentelli [23], pp. 147–161
19. Behrmann, G., Hune, T., Vaandrager, F.W.: Distributing timed model checking - how the search order matters. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 216–231. Springer, Heidelberg (2000)
20. Behrmann, G., Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Efficient timed reachability analysis using clock difference diagrams. In: Halbwachs, Peled [43], pp. 341–353
21. Behrmann, G., Larsen, K.G., Pelánek, R.: To store or not to store. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 433–445. Springer, Heidelberg (2003)
22. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Optimal scheduling using priced timed automata. SIGMETRICS Performance Evaluation Review 32(4), 34–40 (2005)
23. Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.): HSCC 2001. LNCS, vol. 2034. Springer, Heidelberg (2001)
24. Bengtsson, J., Yi, W.: On clock difference constraints and termination in reachability analysis of timed automata. In: Dong, J.S., Woodcock, J. (eds.) ICFEM 2003. LNCS, vol. 2885, pp. 491–503. Springer, Heidelberg (2003)
25. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)

26. Bérard, B., Petit, A., Diekert, V., Gastin, P.: Characterization of the expressive power of silent transitions in timed automata. Fundam. Inform. 36(2-3), 145–182 (1998)
27. Bouyer, P.: Untameable timed automata! In: Alt, H., Habib, M. (eds.) STACS 2003. LNCS, vol. 2607, pp. 620–631. Springer, Heidelberg (2003)
28. Bouyer, P., Brihaye, T., Markey, N.: Improved undecidability results on weighted timed automata. Inf. Process. Lett. 98(5), 188–194 (2006)
29. Bouyer, P., Brinksma, E., Larsen, K.G.: Staying alive as cheaply as possible. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 203–218. Springer, Heidelberg (2004)
30. Bouyer, P., Cassez, F., Fleury, E., Larsen, K.G.: Optimal strategies in priced timed game automata. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 148–160. Springer, Heidelberg (2004)
31. Bouyer, P., Larsen, K.G., Markey, N., Rasmussen, J.I.: Almost optimal strategies in one clock priced timed games. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 345–356. Springer, Heidelberg (2006)
32. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: A model-checking tool for real-time systems. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 546–550. Springer, Heidelberg (1998)
33. Brihaye, T., Bruyère, V., Raskin, J.-F.: On optimal timed strategies. In: Pettersson, Yi [66], pp. 49–64
34. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 66–80. Springer, Heidelberg (2005)
35. Cassez, F., David, A., Larsen, K.G., Lime, D., Raskin, J.-F.: Timed control with observation based and stuttering invariant strategies. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 192–206. Springer, Heidelberg (2007)
36. Čerāns, K.: Decidability of bisimulation equivalences for parallel timer processes. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 302–315. Springer, Heidelberg (1993)
37. Courcoubetis, C., Yannakakis, M.: Minimum and maximum delay problems in real-time systems. In: Larsen, Skou [60], pp. 399–409
38. D'Argenio, P.R., Katoen, J.-P., Ruys, T.C., Tretmans, J.: The bounded retransmission protocol must be on time! In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 416–431. Springer, Heidelberg (1997)
39. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
40. Ernits, J.P.: Memory arbiter synthesis and verification for a radar memory interface card. Nord. J. Comput. 12(2), 68–88 (2005)
41. Fahrenberg, U., Larsen, K.G.: Discount-optimal infinite runs in priced timed automata. Electr. Notes Theor. Comput. Sci. 239, 179–191 (2009)
42. Fehnker, A.: Scheduling a steel plant with timed automata. In: RTCSA, pp. 280–286. IEEE Computer Society, Los Alamitos (1999)
43. Halbwachs, N., Peled, D.A. (eds.): CAV 1999. LNCS, vol. 1633. Springer, Heidelberg (1999)
44. Hansen, M.R., Madsen, J., Brekling, A.W.: Semantics and verification of a language for modelling hardware architectures. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 300–319. Springer, Heidelberg (2007)

45. Hendriks, M.: Model checking the time to reach agreement. In: Pettersson, Yi [66], pp. 98–111
46. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Inf. Comput. 111(2), 193–244 (1994)
47. Hune, T., Larsen, K.G., Pettersson, P.: Guided synthesis of control programs using uppaal. Nord. J. Comput. 8(1), 43–64 (2001)
48. Jensen, H.E., Larsen, K.G., Skou, A.: Scaling up Uppaal automatic verification of real-time systems using compositionality and abstraction. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 19–30. Springer, Heidelberg (2000)
49. Jensen, K., Podelski, A. (eds.): TACAS 2004. LNCS, vol. 2988. Springer, Heidelberg (2004)
50. Jessen, J.J., Rasmussen, J.I., Larsen, K.G., David, A.: Guided controller synthesis for climate controller using uppaal tiga. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 227–240. Springer, Heidelberg (2007)
51. Karp, R.M.: A characterization of the minimum cycle mean in a digraph. Disc. Math. 23(3), 309–311 (1978)
52. Lamport, L.: Real-time model checking is really simple. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 162–175. Springer, Heidelberg (2005)
53. Larsen, K.G., Behrmann, G., Brinksma, E., Fehnker, A., Hune, T., Pettersson, P., Romijn, J.: As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 493–505. Springer, Heidelberg (2001)
54. Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Efficient verification of real-time systems: compact data structure and state-space reduction. In: IEEE Real-Time Systems Symposium, pp. 14–24. IEEE Computer Society, Los Alamitos (1997)
55. Larsen, K.G., Mikucionis, M., Nielsen, B., Skou, A.: Testing real-time embedded software using uppaal-tron: an industrial case study. In: Wolf, W. (ed.) EMSOFT, pp. 299–306. ACM, New York (2005)
56. Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Clock difference diagrams. Nord. J. Comput. 6(3), 271–298 (1999)
57. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. STTT 1(1-2), 134–152 (1997)
58. Larsen, K.G., Rasmussen, J.I.: Optimal conditional reachability for multi-priced timed automata. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 234–249. Springer, Heidelberg (2005)
59. Larsen, K.G., Rasmussen, J.I.: Optimal reachability for multi-priced timed automata. Theor. Comput. Sci. 390(2-3), 197–213 (2008)
60. Larsen, K.G., Skou, A. (eds.): CAV 1991. LNCS, vol. 575. Springer, Heidelberg (1992)
61. Lindahl, M., Pettersson, P., Yi, W.: Formal design and analysis of a gear controller. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 281–297. Springer, Heidelberg (1998)
62. Maler, O.: Timed automata as an underlying model for planning and scheduling. In: Fox, M., Coddington, A.M. (eds.) AIPS Workshop on Planning for Temporal Domains, pp. 67–70 (2002)
63. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems (an extended abstract). In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 229–242. Springer, Heidelberg (1995)
64. Møller, J.B., Lichtenberg, J., Andersen, H.R., Hulgaard, H.: Difference decision diagrams. In: Flum, J., Rodríguez-Artalejo, M. (eds.) CSL 1999. LNCS, vol. 1683, pp. 111–125. Springer, Heidelberg (1999)

65. Ouaknine, J., Worrell, J.: Universality and language inclusion for open and closed timed automata. In: Maler, O., Pnueli, A. (eds.) HSCC 2003. LNCS, vol. 2623, pp. 375–388. Springer, Heidelberg (2003)
66. Pettersson, P., Yi, W. (eds.): FORMATS 2005. LNCS, vol. 3829. Springer, Heidelberg (2005)
67. Rasmussen, J.I., Larsen, K.G., Subramani, K.: Resource-optimal scheduling using priced timed automata. In: Jensen, Podelski [49], pp. 220–235
68. Tripakis, S., Altisen, K.: On-the-fly controller synthesis for discrete and dense-time systems. In: Wing, J.M., Woodcock, J.C.P., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 233–252. Springer, Heidelberg (1999)
69. Yi, W., Pettersson, P., Daniels, M.: Automatic verification of real-time communicating systems by constraint-solving. In: Hogrefe, D., Leue, S. (eds.) FORTE. IFIP Conference Proceedings, vol. 6, pp. 243–258. Chapman & Hall, Boca Raton (1994)

rCOS: Theory and Tool for Component-Based Model Driven Development[*]

Zhiming Liu, Charles Morisset, and Volker Stolz

Intl. Institute for Software Technology (UNU-IIST),
United Nations University, Macao SAR, China
{lzm,morisset,vs}@iist.unu.edu
http://rcos.iist.unu.edu

Abstract. We present the roadmap of the development of the rCOS theory and its tool support for component-based model driven software development (CB-MDD). First the motivation for using CB-MDD, its needs for a theoretical foundation and tool support are discussed, followed by a discussion of the concepts, techniques and design decisions in the research of the theory and the development of the prototype tool. The concepts, techniques and decisions discussed here have been formalized and published. References to those publications are provided with explanations. Based on the initial experiences with a case study and the preliminary rCOS tool development, further development trajectory leading to further integration with transformation and analysis plug-ins is delineated.

Keywords: contract, component, design pattern, model transformation.

1 Introduction

Complexity has long been recognized as an *essential property* of software, not an *accidental one* [12,13]. The inherent complexity is due to four fundamental attributes; the *complexity of the domain application*, the *difficulty of managing the development process*, the *flexibility possible* to offer through software, and the *problem of characterizing the behavior* of software systems [1]. The first attribute implies the challenges in the requirements capture and analysis and the problem of *changeability* of software to meet continuously changing requirements for additional functionality and features, the second one focus on the management of the development process and team, the third indicates the difficulties and creativity needed in making the right design decisions, and the final one pin-points the difficulty in software *analysis, validation* and *verification* for *correctness assurance.*

We are now facing an even greater scale of complexity with modern *software-intensive systems* [41]. We see these systems in our everyday life, such as in

[*] Supported by the projects HighQSoftD and HTTS funded Macao S&TD Fund and the grants CNSF No. 60970031, NSFC No.90718014 and STCSM No.08510700300.

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 62–80, 2010.

aircraft, cars, banks and supermarkets [5]. These systems provide their users with a large variety of *services* and *features*. They are becoming increasingly *distributed*, *dynamic* and *mobile*. Their components are *deployed* over large networks of *heterogeneous platforms* and thus the *interoperability* of the distributed components becomes important. Components are also *embedded* within hardware devices. In addition to the complexity of functional structure and behavior, modern software systems have complex aspects concerning *organizational structure* (i.e. *system topology*), *distribution*, *interactions*, *security* and *real-time*.

For example, the CoCoME benchmark system [5] is a *trading system* for an enterprise of stores. It has components for *processing sales*, *management of different stores* and *management of enterprise*. These components are deployed on different computers in different places, and they interact among themselves and with external systems such as banks and product suppliers, through different middlewares (such as RMI and CORBA). The application software components have to be embedded with controllers of hardware devices such as product code scanners, printers, and credit card readers.

A complex system is open to total breakdown [35], and we suffer from the long lasting *software crisis*[1] where projects fail due to our failure to master the complexity. Given that the global economy, as well as our every day life, depends on software systems, we cannot give up in advancing the theory and the engineering methods to master the increasing complexity of software development.

In this paper, we present the rCOS approach to CB-MDD for handling software complexity. Section 2 motivates the research in the rCOS modeling theory and the development of a prototype tool. There, we first show how CB-MDD is a natural path in the advance of software engineering in handling complexity by separation of concerns. It is then followed by a discussion of the key theme, principles, challenges, and essential techniques of a component-based model driven approach. We stress the importance and the add-ins of *tool support*, and argue that a tool must be supported by a sound and appropriate *modeling theory*. In Section 3, we summarize the theoretical aspects of rCOS and show how they meet the needs discussed in Section 2. Section 4 reports our initial experience with the tool, and based on the lessons learned, we delineate the further development trajectory leading to further integration with transformation and analysis plug-ins. Concluding remarks are given in Section 5.

2 Natural Path to CB-MDD

In software engineering, the complexity of a system development is mastered by *separation of concerns*. Models of software development processes are proposed for dividing and conquering the problems in software development. The traditional approaches are mostly variants of the *waterfall model* in that problems of software development are divided into the problems of *requirements capture and analysis*, *design*, *coding* and *testing*, and solved at different level of abstractions.

[1] Booch even calls this state of affairs “normal” [1].

2.1 Early Notions of Components and Models

In a waterfall process, the principles of *structured programming* [10] and *modularization* [34] are used to construct a software system by decomposing it into *procedures* and *modules*. The concept of modules is thus an early analogy of the notion of *components*. With the notions of decomposition and modules, the initial waterfall process changed into the *evolutionary development*, and the *spiral model* is proposed with more consideration of project management and risks control [38]. Procedures as components do not support large scale software development, and the original modules as components do not have *explicitly specified contracts of interfaces* to be used in third party composition [40].

For applications with higher demands for correctness assurance and dependability, rigorous testing after implementation for *software defect detection* is not enough, and a best effort at defect detection is required in each phase of the development process. This advances the waterfall model to the *V-model*. In a V-model development process, *artifacts* produced in each phase should be properly documented and validated. *Tools* are then developed to help in the documentation of the artifacts produced in different cycles for different versions to ensure their consistency. Tools for prototyping and simulation are also used for system validation in different phases. Taking documents of artifacts as “models”, though not necessarily formal models, a non-trivial software development is to produce, validate, and test models with some tool support.

Software systems for safety critical applications are required to be *provably correct*, not only syntactically but also semantically . For this, each phase is required to produce *semantically verifiable models* and this needs the *modeling notation to be formally defined* and a *sound theory* to be developed for verifying and reasoning about properties of models. Model verification is only realistic and trustworthy with support of automated tools. Indeed, in the last forty years or so, a large body of formal notations and theories have been developed based on two different models, *state based* [21,42,37], *event based* [18,30] and *property based* [28,29,22]. Development of *theorem proving* [39], *model checking* [20,36] and *simulation* [9] tools have rapidly advanced recently. All of these methods and techniques have their uses in some aspects of system development, and the challenge is now to select and adapt them to a harmonic whole. It is important to note that development of tool support also requires the models formally described because tools are software and can only manipulate formally defined notations.

2.2 Theoretical and Tool Support to Successful CB-MDD

Components and models are in the scope of software engineering. In industrial practice models are often manually built, and an initial code outline is generated from a model of a detailed design. Only recently component-based model driven design is becoming a clear and mainstream discipline. The discipline requires that in a development process

- each phase is based on the *construction of models*,
- models in later phases are constructed from those in earlier phases by *model transformations*,
- code is an executable model generated from models in the design phase.

For a safety critical application, it is further required that

- the models constructed are verifiable,
- model transformation are proven to preserve desirable properties of the models, such as functional correctness, and
- the model transformations generate conditions about the models, called *proof obligations*, that are verified by using verification techniques and tools.

This implies that what is critical to CB-MDD is a modeling approach with sound *theoretical foundation* and strong *technical and tool support*. The approach should integrate techniques and tools for *correct by construction* through model transformations and those for *analysis* and *verification*. The transformations support the engineering design and reduce the burden on automated verification. These form the theme of the rCOS method and in what follows we describe the key features of this modeling approach.

Multi-dimensional separation of concerns. The models at each phase should separate the *different views* and characterize the different aspects of the software *architecture* at the given level of abstraction. A unified set of different notations, such as UML, is often used as the modeling language. There is also a consensus on notations for the different views of a software system, such as use cases for requirements, class diagrams for structural design, sequence and state diagrams for interaction protocols and reactive behaviors. Separation of concerns[2] has to deal with the important issue of the *correctness* and *consistency* of *different views* [4,33]. There are existing UML profiles with precisely defined syntaxes and tools for building models, which ensure and check their syntactic correctness and consistency. The problem of semantic correctness and consistency is much harder, and a semantic theory has to be developed for the unified modeling language. Our experience with rCOS [3,6] is that the challenge lies in that the semantic theory must support separation of concerns to allow us to factor the system model into models of different views and to consistently integrate models together under an execution semantics of the whole system. This is even more difficult when we have to integrate *object-orientation* into a theory of CB-MDD such as rCOS [17]. The semantic theory is also needed for the models to be verifiable by verification tools and *manipulatable* by automated correctness preserving model transformations, and properly formalized models are needed for automatic generation of test suites. Separation of concerns and multi-view modeling are the key to the scalability of the method and allow us to take advantage of different theories and their tool support for the analysis and manipulation of different views and aspects. To this end,

[2] Here, multi-dimensional separation of concerns is related to what it meant in [32], but with a wider extension.

we need to advance the ideas of "putting theories together" [2,15] and "unifying semantic theories of programming" [19] to produce, analyze and transform models with integrated tool support.

Object-orientation in CB-MDD. The rCOS method integrates object orientation as an important part. Among other reasons including reusability and maintenance, there are three reasons from our own experience. Firstly, object-oriented analysis and design complements structured analysis and design in handling the complexity of the organizational structure of the system [1]. *Concepts*, their *instances* and their *relations* in the application domain naturally form the *structure of the domain* and can be directly modeled by the notions of *classes*, *objects* and *associations* or *attributes*. This is found to be very useful for domain understanding, and requirements capture and analysis. The combination of use cases for functional requirements analysis and decomposition with object orientation for structural analysis and decomposition works systematically and effectively in both practical and formal component-based development [23,6]. Secondly, the use of design patterns [14,23] makes the object-oriented design through model transformations more systematic and thus has much higher possibility for automation [17,27]. Finally, most, if not all, industrial component-based technologies are implemented in the object-oriented paradigm.

Component-based architecture. For a model driven design, we need a precise and strong notion of *component-based architecture* such that

1. it describes the system functionalities and behavior,
2. it captures the decomposition of the system into *components* that *cooperate* and *interact* to fulfill the overall system functionalities,
3. it supports *composition, coordination* and *connection* of components and describes the *hierarchical* and *dependency* relationships among *components.*

Unlike the conventional notions of components which used to be about programs in code, components in CB-MMD are involved in all phases of the development and represented in different languages. Composition, coordination and connection of components can only be defined based on the *interface behaviors* of the components. Independent deployment, interoperability, reuse of existing (commercial off-the-shelf) components also require components to have explicitly specified *contracts of interfaces* [40,3,6]. The modeling language should be expressive enough for specifying the multi-view and hierarchical nature of components and allows *abstraction by information hiding.*

Scaling and automating refinement. Formal techniques and tools for verification are mainly for defect detection. They do not support decision making of the designer in systematic and correct model construction. The basic notion we find in formal methods that supports correct by design is *program refinement* [31,19]. However, the classical refinement techniques need to be generalized and unified to support the separation of concerns and allow models of different views to be refined separately and hierarchically, such as data *functionality refinement*, *interaction refinement* and *object-oriented structure refinement* [17,3,44]. We need to scale up refinement rules and automate them via exploration of formal use [17,27] and automation of design patterns [14,23] for abstract models and refactoring rules [11] for models at lower levels of abstraction, such as a design class diagram. Automation of the scaled refinement rules provides us with the implementation of model transformations. Application of such a model transformation generates *proof obligations* for verification of properties of the models before and after the transformation. These verification tasks can be carried out by integrated verification tools, such as a theorem prover, a static checker, or a model checker. This gives a seamless combination of tools for verification and correctness by construction. Therefore, automation of a rich set of refinement rules is the key to extensively tool supported model driven development. It turns out to be the greatest challenge, too, in the tool development.

Tool supported development process. A convincing conclusion drawn from the above discussion is that CB-MMD needs an integrated tool suite supported by a sound theory, instead of a single purpose tool. With the tool support, a software system must be developed in a clearly defined engineering process in which different activities at different stages of development are performed by project participants in different roles. We take this view very important, as only with an engineering process it allows us to define at which points in the development process should various models (or informally called *artifacts*) be *produced*, and different kinds of *manipulation*, *analysis*, *checking* and *verification* be performed, with different tools. The rCOS tool, introduced in Sec. 4, intends to provide such an integrated tool suite and already allows one to run some model transformations and checking.

3 Theoretical Foundation of rCOS

We summarize the main concepts and models of software artifacts defined in rCOS without going into technical details. Such details can be found in our earlier publications [17,3,6,44].

An essential concept in rCOS is that of *components* and rCOS provides a multi-dimensional approach to modeling a component. Along the *vertical dimension*, a component K has various models with different details, i.e. in different levels of abstraction, in different stages and for different purposes. A *component implementation* as a piece of program, *contracts* of components at the level of requirements specification and design, and *publications* for component usages and synthesis. On the horizontal dimension is the *hierarchical* and *dependency* relationships among *components*. It captures the composition of the component from sub-components with *connectors*, *coordinators* and *glue programs*. The third dimension separates the *different views* and characterize the different aspects of the component at the given level of abstraction, including the data and class structure, data functionality, interaction protocol, reactive behavior, etc.

3.1 Component Implementation and Component Refinement

At the code level, a component has a *provided interface $K.pIF$*, possibly a *required interface $K.rIF$* and a piece of program code $K.code(m)$ for each method $m()$ in the provided interface. The required interface $K.rIF$ contains the methods that are called in the code of the component K, but not declared in the provided interface or defined as internal methods in the component.

Interfaces. In rCOS, an interface is a *syntactic notion*, and each *interface I* is a declaration of a set *$I.fields$* of typed variables of the form $x : T$, called *fields*, and a set *$I.methods$* of method signatures of the form $m(x : T, y : V)$, where $x : T$ and $y : V$ are the input and output parameters with their types.

UTP as root of semantic theory. In principle, different components can be implemented in different programming languages. We thus need a semantic model for "unifying theories" of different programming languages, and thanks to Hoare and He, we use the well studied UTP [19]. The essential theme of UTP that helps rCOS is that *a program in any programming language can be defined as a predicate*, called a *design*. A *design* is specified as a pair of pre- and post-conditions, denoted as $pre(x) \vdash post(x, x')$, of the *observables* x and x' of the program, and its says that if the program executes from a state where the *initial values* x satisfies $pre(x)$ the programs will terminates in a state where the final values x' satisfies the relation $post(x, x')$ with the initial values x. Observables include program variables and auxiliary variables dependable on the observable behavior being defined, such as termination, denoted by ok and ok' in sequential programs and interaction traces tr and tr' in communicating programs. The rCOS tool provides a textual language for specifying the methods, with a syntax mixing both designs (pre/post-conditions, non-deterministic choice, etc) and sequential code (conditional statement, loop, etc).

Semantics and refinement of components. With the definition of designs and the refinement calculus established in UTP, the semantics of a component K is defined as a function $\lambda C_{rIF} \cdot spc.K$. The type of the function is from the set of specification functions C of the required method methods in $K.rIF$ to the specification functions of the required methods in $K.pIF$. For any specification function C (called a *contract* later) of the required interface $K.rIF$, $spc.K(C)$ is the function that gives each required method $n()$ a design $C(n())$ calculated in the calculus of UTP, and thus defines the semantics of each $m()$ of the provided interface $K.pIF$ as a design. The semantics $spc.K(C)(m)$ is calculated from the code of m by replacing each invocation to a required method $n()$ by the semantics $C(n())$. Notice that if $K.rIF$ is an empty interface, $\lambda C_{rIF} \cdot spc.K$ is a constant function.

Components are in general reactive programs and thus concurrent programming languages are used for their implementation. The semantics of each method is thus defined as a *reactive design*. In [3], the *domain of reactive designs $\mathcal{RD}$* is a sub-domain of the domain of designs $\mathcal{D}$ characterized by *lifting function* $\mathcal{H} : \mathcal{D} \rightarrow \mathcal{RD}$ such that $\mathcal{H}(p \vdash R) \mathrel{\widehat=} (true \vdash wait') \triangleleft wait \triangleright (p \vdash R)$, stating that when *wait* is *true* the execution stays in the *wait* state and proceeds according to the design $p \vdash R$ otherwise. In this specification, Boolean observables *wait* and *wait′* represents the synchronization so that the execution of the program is suspended when it is in a *wait* state. We also introduce *guarded designs* $g\&(p \vdash R)$ to specify the reactive behavior $\mathcal{H}(p \vdash R) \lhd g \rhd (true \vdash wait')$, where "$... \lhd g \rhd ...$" is the math infix operator operators for the programming construct **if** g **then** ... **else** ....

A component K_1 is a *refinement* of K, denoted by $K \sqsubseteq K_1$, if they have the same provided and required interfaces, and for each contract C of $K.rIF$ and each provided method $m \in K.pIF$, the design $spc.K(C)(m) \sqsubseteq spc.K_1(C)(m)$ holds in the refinement calculus of UTP. A work in progress is to automatically generate the proof obligations in Isabelle/HOL that a design is a refinement of another one.

Object-orientation in rCOS. To support object-oriented design of components, types of fields of component interfaces can be *classes* and thus the values of these fields are objects. We have extended UTP to define object-oriented programs and developed an object-oriented refinement calculus to handle both structure and behavior refinement [17,44]. The object-oriented semantic model in rCOS provides formal treatment of *aliasing*, *inheritance* and *dynamic binding*. These features are needed in CB-MDD when constructing, transforming and verifying models in later stages of the development.

3.2 Contracts

In the CB-MDD paradigm, a component is developed by model transformations from its *requirements analysis model* to a *design model* and finally an *implementation model*. We take the view that the analysis model specifies the functionalities from the users' perspective and describes *what* does the component do for *what kind* of users. A kind of users is called an *actor* in the UML community and the actors together define the *environment* of the component.

Component-based development allows the use of an existing component to realize a model of a component in the analysis model. The *fitness* of the existing component for the purpose in the model must be checkable without information about the design and implementation of the existing component. For this, the analysis model of a component should be a black box characterization of what is needed for the component to be designed and used in building and maintaining a software. The information needed depends on the application of the component. For example, for a sequential program, the specification of the static data functionality of the interface methods is enough, but information about the worst execution time of methods is needed for a real-time application, and for reactive systems we also need reactive behavior of the component and the interaction protocol in which the environment interacts with the component. For the treatment of different applications, the intention of rCOS is to support incremental modeling and separation of concern.

Contracts for black-box modeling. In rCOS, a black box behavior model of interfaces called a *contract* is defined, and its current version focuses on reactive systems. A contract $C = (I, \theta, \mathcal{F})$ defines for an interface I, denoted by $C.IF$,

- an *initial condition* θ, denoted by *C.init*, that specifies the allowable initial states of the intended component,
- a specification function $\mathcal{F}$, and denoted by *C.spec*, specifying the reactive behavior by giving each method $m \in$ *C.IF.methods* a reactive design *C.spec*(m).

Note that the specification function *C.spec* combines the static (data) functionality view and the reactive dynamic behavior view [4]. A contract also has a third view, the *structure view*, that defines the data- and class structures. It is represented by a UML *class diagram*it, which defines the data types and classes of the objects of the component.

Refinement of contracts. For the study of the consistency of contracts, separation of concerns and refinement among contracts, an *execution semantics* of a contract C (and thus components) is defined in [3] by its *failures*, *failure.C*, and its *divergences*, *divergence.C* [36]. A contract C_1 *refines* a contract C_2, denoted by $C_2 \sqsubseteq C_1$, if C_1 is neither more likely to diverge, i.e. $\mathit{divergence}.C_1 \subseteq \mathit{divergence}.C_2$, nor more likely to block the actors, i.e. $\mathit{failure}.C_1 \subseteq \mathit{failure}.C_2$.

Correctness of components. It is now clear that $\lambda C_{rIF} \cdot \mathit{spc}.K$ calculates a contract of the provided interface of K for a given contract of C_r of its required interface. A component K *fulfills* or *implements* a contract C if there exists a contract such that $C \sqsubseteq \mathit{spc}.K(C_r)$. Clearly, $\mathit{spc}.K(C_r)$ is the **s**trongest **p**rovided **c**ontract for C_r. We call a pair of contracts $C = (P, R)$ of *K.pIF* and *K.rIF* a *contract of component* K if $P \sqsubseteq \mathit{spc}.K(R)$, that is K fulfills the provided services P if the environment provides K with services R. We define the relation of *alternate refinement* between component contracts such that for publications $C_1 = (P_1, R_1)$ and $C_2 = (P_2, R_2)$, $C_1 \sqsubseteq C_2$ if $P_1 \sqsubseteq P_2$ but $R_1 \sqsupseteq R_2$, meaning that the refined component provides "better" services while requiring "weaker" services.

3.3 Publications

A contract of the interface of a component is good to be used by the designer of the component and for verification of the correctness of the design. However, it is rather operational for a user of the component. A user prefers a more declarative and more abstract specification in the form of a user manual. This is the notion of *publications* [16,25] for assembling components. Therefore, a publication is about the usage of the component.

In rCOS, a *publication* $P = (I, \theta, \mathcal{S}, \mathcal{T})$ specifies for an interface I, denoted by *P.IF*,

- an *initial condition* θ, denoted by *P.init*, that specifies the allowable initial states of the intended component,
- a specification function $\mathcal{S}$, and denoted by *P.spec*, specifying the *static data functionality* by giving each method $m \in$ *P.IF.methods* a design *C.spec*(m) (without guard), and

– a *protocol* $\mathcal{T}$, denoted by *P.prot*, that is a set of traces over the method names of *P.IF.methods*, specify the assumed *work flows* or *interaction protocol* in which the actors use services of the intended component.

We define that for a same interface I, a publication P_2 is a *refinement* of a publication P_1 of I if $P_2.init \Rightarrow P_1.init$, $P_1.prot \subseteq P_2.prot$ and for each interface methods m, $P_1.spec(m) \sqsubseteq P_2.spec(m)$.

In [25], a function $\mathcal{P}$ is defined to obtain a publication from a contract C, and a function $\mathcal{C}$ to obtain a contract from a publication; and the pair $\mathcal{C}, \mathcal{P}$ forms a *Galois Connection* between the domains of contracts and publications with respect to the refinement partial orders.

Component publications and their faithfulness. A *publication* of a component is a specification $U = (G, A)$, where G and A are publications of the provided interface *K.pIF* and required interface *K.rIF*, respectively. We define the relation of *alternate refinement* between component publications such that for publications $U_1 = (G_1, A_1)$ and $U_2 = (G_2, A_2)$, $U_1 \sqsubseteq U_2$ if $G_1 \sqsubseteq G_2$ but $A_1 \sqsupseteq A_2$.

We now extend the functions $\mathcal{P}$ and $\mathcal{C}$ to publications and contracts of components, i.e. to pairs of of publications and pairs of contracts, respectively. A publication $U = (G, A)$ of K is *faithful* if there is a contract $C = (P, R)$ of K such that $U \sqsubseteq \mathcal{P}(C)$, i.e. $G \sqsubseteq \mathcal{P}(P)$ and $A \sqsupseteq \mathcal{P}(R)$. This is the basis for *publication certification.* A component K with a faithful publication U *fits* in the position of contract $C = (P, R)$ in a model, if U refines $\mathcal{P}(C)$. In [25], a mapping $\mathcal{C}$ from publications to contracts is also defined and a theorem is proven that $(\mathcal{P}, \mathcal{C})$ forms a *Galois connection* between the domain of contracts and the domain of publications.

A component K with a publication $U = (G, A)$ is *substitutable* by a component K_1 with a publication $U_1 = (G_1, A_1)$ if $U \sqsubseteq U_1$ that is defined as $G \sqsubseteq G_1$ and $A \sqsupseteq A_1$. Notice that contracts and publications of a component are truly *black box specifications* of the component. Contracts are used for design and verification of components while publications are used for substitutability and assembling.

Theorem of separation of concerns. The fact that $(\mathcal{P}, \mathcal{C})$ forms a *Galois connection* between the domain of contracts and the domain of publications makes the rCOS theory support the separation of concerns. It allows to preserve the faithfulness of a publication by refining the the data functionality and shrinking the protocol in of the provided publication while weakening the data function functionality and enlarge the protocol of the required required publication.

The notions of contracts and publications of an interface can be combined to a notion of *extended contract*[3] $C = (I, \theta, \mathcal{F}, \mathcal{T})$ which specifies the interface, the initial condition, reactive designs (behavior) and interaction protocol of an intended component. The protocol and the reactive behavior are therefore required to be consistent so that all traces in the protocol are allowed by the reactive behavior $\mathcal{F}$. Our experience with the CoCoME example [5] shows it is desirable for the design to specify a use case as an extended contract of the provided interface of a component (see Section 4 too).

[3] This is actually the notion of contract defined in [17].

The theorem of separation of concerns in [3] allows to refine the static data functionality and reactive behavior of a contract separately to preserve the consistency of an extended contract, and thus the faithfulness of a publication. It is interesting to point out that object-oriented refinement [17,44] makes formal use of design patterns. It is thus crucially important for the refinement of the static functionality and for scaling and automating refinement to develop tool support.

3.4 Composition

The notion of *composition* is essential for a component-based design and must be defined for models of components at all levels of abstraction, and it should be consistently refactorable to composition of interfaces, static functionality, reactive behaviors and interaction protocols. As a *component-based architecture description langauge*, rCOS defines the basic composition operators for *renaming* interface methods, *restriction* on the environment from access to provided methods, *internalization* of provided methods to make them autonomously executed when they are enabled, *plugging* the provided interface of one component to the required interface of another components, and *disjoint parallel composition* of components. Internalization and plugging together can represent general components coordination. In the following, we discuss the nature of these composition operators at different levels of abstraction.

Composition of contracts and publications of components. The definitions of *renaming*, *restriction*, *plugging* and *disjoint parallel compositions* for contracts and publications are relatively easier than *internalization* [25]. However, for the plugging composition is conditional. A contract $C_1 = (P_1, R_1)$ (or a publication $P_1 = (G_1, A_1)$) is *composable* with $C_2 = (P_2, R_2)$ (resp. publication $P_2 = (G_2, A_2)$) if the provided contract P_1 in C_1 (resp. G_1 in P_1) refines the required contract R_2 in C_2 (resp. A_2 in P_2).

The difficulty in defining internalization is first to make sure the result of a composition is still a contract defined in rCOS. For this, the effects of the autonomous internal executions of the internalized methods must be aggregated into the remaining interface methods. In [3] a definition for internalization (called synchronization there) of a component by a *process* is given and the result proven to be a component. However, this is better used at the implementation level. In [25], internalization is directly defined for contracts and publications.

Composition of component implementations. The composition operators *renaming*, *restriction*, *plugging* and *disjoint parallel composition* are implemented as simple *connectors* following the semantics defined in [3,25]. Internalization is implemented by using *processes* (think of a scheduling process) that automatically calls the internalized methods for execution when they are enabled (i.e. their guards become true). The semantics of the synchronous composition of a component and a process is defined in [3] and there the composed entity is proven to be a component.

Compositional modeling, refinement and verification. At all levels of abstraction, the composition operations are proven to be monotonic with respect to the refinement order. This allows us to carry out compositional design by model transformations. The relationships of fulfillment of contracts by components, faithfulness of component publications, and the fitness of a component in a model are preserved by the composition operations (for composable compositions). This enables us to do compositional analysis, verification and certification.

4 The rCOS Tool

The rCOS tool focuses on CB-MDD and is oriented towards organizing the development activities. It introduces a body of concepts and a hierarchy of artifact repositories, designed to support team collaboration on development of the models and generation of code (cf. the paragraph on **tool supported development process** in Section 2.2). At the top-level of component repositories is the *application workspace*, representing the whole modelling and development space of an application. The application workspace is partitioned into *components* through hierarchical use cases. A component is characterized by its subset in the model of different views and represented in different forms depending on the phases of the development. The application maintains the (*requirements*) *analysis model*, the *design model* and the *platform specific design and implementation*. The informal requirements document is mainly a description of the use cases and their relationships. It is represented in a structured natural language (not stored in the model) and the use case diagrams [6]. Use cases may *refer* or *use* to other use cases, hence the hierarchical notion of sub-use cases. *Each use case is called a component at this level.*

To support construction, understanding and transformations of these models, a *UML profile* is defined for rCOS, and models of the views of reactive behavior, interaction and class structure are created as instances of the metamodel of the UML profile [7]. The UML model has an equivalent representation in the rCOS textual syntax and is the input for the various formal analyses.

4.1 Tool Support to Requirement Analysis

An *Analyst* works on a component, that is, a use case of the application, by studying its textual requirements and the application domain, and creates an *analysis model* consisting of a *use case diagram*, a *conceptual class diagram*, an *interface sequence diagram*, the *functionality specification* of the interface methods, and a *state machine diagram*. The use case diagram represents the dependency relation between the actors in this use case and referenced use cases.

Models of different views. The use case diagram describes the dependency relationships between the actors and the component. Some actors are components external to this component. The conceptual class model represent the domain concepts and objects with their structural inheritance relations involved in the use cases of the use case diagram. Methods are designed for the component interface, but not for the other conceptual classes at this stage. The interface sequence diagram models the interactions between the actors and the component, and the interactions among the subcomponents corresponding to the sub-use cases of the use case diagram. It is thus in general a *component sequence diagram* (cf. the next subsection). The state diagram represents reactive behavior of the component and characterizes the flow of control and synchronization of the component. The functionality specification of the interface methods specifies the pre- and post-conditions of the methods in rCOS. Therefore, the different views of a use case together form a model of an extended contract of a component whose provided interface provides the methods for the actors to call.

Analysis and validation. The Analyst is responsible for verifying that the models of the different views are consistent, and validating it against the informal requirements document. The syntactical consistency checking is implemented as part of the type checker of rCOS, though the full implementation is still ongoing.

We have developed a prototype of a tool for automatic protyping from an analysis model [24] for validating requirements. For the dynamic consistency of the sequence diagram and state machine diagram, we translate them into CSP processes and check *deadlock freedom* of their composition with the CSP model checker FDR2 [8]. Here, consistency means that all interaction scenarios defined by the sequence diagram are accepted by the state machine. Likewise, we check *faithfulness of the contract* with regard to an executable rCOS specification (the component does not deadlock for any interaction in the contract).

Application dependent properties, such as safety and liveness, can be verified by a combination of model checking the CSP process of the state diagram and static analysis of the functionality specification of the interface methods.

The Analyst may iterate over this model, creating, decomposing and refining models. It may also be necessary to revise the informal requirements documents according to the results of the analysis and validation. She can declare a dependency on another component and, if the component depends on other components, the Analyst specifies which interface these *required components* have to provide, or she may introduce abstract models of the required components. A verified and validated model can be *frozen* and is used for the design of components by a *Designer*.

Remarks. With the support of the tool, the syntactic consistency can be guaranteed, such as the method names, parameters and name of attributes in different views. The construction of the class diagram, sequence diagram and state diagram is fully supported by the tool. It however needs domain experts who understand the syntax and its semantics for writing the correct pre- and post-conditions of the methods. Another difficulty arises in a team where with multiple analysts working on different components (even initially disjoint components). Decomposition of a use case component requires the awareness of the progress being made on other components to avoid duplicate introduction of components and to accommodate changes obtained through analysis and design of other components. There seems to be no formal and systematic tool support to ensure across component consistency except for having project review meetings to decide what changes should be made. Our experience from the CoCoME case study [5] is that modelers of different components have to spend a lot time to discuss with each other the models that they are working on and informing each other about any new components they introduce.

4.2 Model Transformation Tool to Support Design

A *Designer* produces a *design model* from an analyzed component model by a sequence of model transforms. In rCOS, we intend to support threes kinds of model transformations for producing an *object-oriented design model* of the component, a *component-based design model* from the object-oriented design model, and a *platform specific design model.*

Object-oriented design of a component. This mainly involves stepwise refinement of the data functionality of the interface methods. The driving force for this is repeated applications of the *Expert Pattern* for Assignment of Responsibilities in object-oriented design [23]. The *Expert Pattern* provides a systematic decomposition of the functionality of a method of an object into responsibilities of its related objects, called *information experts*, which *maintain* or *know* the information for carrying out these responsibilities. The related objects of an object o can be defined by the navigation paths from o, and they are derivable from the class diagram (and from the rCOS class declarations).

Formalizing and implementing Expert Pattern. We classify the primitive responsibilities of an object o of class M into *knowing responsibilities* and *doing responsibilities* [23]. Each object is considered to be responsible for knowing its attributes and doing its methods. It is also responsible for knowing its linked objects and for delegating tasks to them, i.e. invoking their methods. For instance, if an object a contains an object b, then a can access to fields and methods of b, but if b contains an object c, then a should not access directly to attributes and methods of c, but rather delegate such actions to b.

Hence, we introduce the following rewriting rules, for any navigation path $p \neq \mathit{this}$ of type M, any fields a and b and any methods m and n:

$$\begin{array}{llll} p.a.b & \longrightarrow p.find\_a\_b() & p.a.m(\bar{x}) & \longrightarrow p.find\_a\_m(\bar{x}) \\ p.m(\bar{x}).a & \longrightarrow p.find\_m\_a(\bar{x}) & p.m(\bar{x_1}).n(\bar{x_2}) & \longrightarrow p.find\_m\_n(\bar{x_1},\bar{x_2}) \end{array}$$

where the following methods are automatically created in the class M when needed:

$$\begin{array}{ll} find\_a\_b()\ \{return\ a.b\}, & find\_a\_m(\bar{p})\ \{return\ a.m(\bar{p})\} \\ find\_m\_a(\bar{p})\ \{return\ m(\bar{p}).a\}, & find\_m\_n(\bar{p},\bar{q})\ \{return\ m(\bar{p}).n(\bar{q})\} \end{array}$$

These rules can be inductively applied to any navigation path containing at least three elements different from *this*, the number of elements in the path being decreased by one at each step.

What a method of an object can do, is to change its own attribute and to delegate the change of the attributes of its linked objects to the corresponding objects. We also introduce a rule concerning the responsibility of objects with respect to the modification of their attributes. For any navigation path $p \neq this$ of type M, any attribute a and any expression e, we introduce the following rule:

$$p.a := e \longrightarrow p.set\_a(e) \text{ with } set\_a(x)\ \{a := x\}$$

At the moment, we have implemented a transformation which takes the full specification of a method $m()$ in normal form and carries out all the responsibility assignments in one go. We plan to implement a transformation that takes a part of a specification designated by the user and carries out one step of the decomposition. For example, she selects a message in a sequence diagram, chooses a sub-expression from the corresponding functionality specification and asks the system to generate the intermediate setters and getters to delegate the accesses. This also generates a more readable design, because it allows the designer to choose meaningful method names.

Other model transformations. Before and after the application of the expert pattern, we can improve the low level design by using other design patterns, such a *High Cohesion* and *Low Coupling* (cf. [23] for informal presentation and [17] for an rCOS formalization) as well as the Creator Patterns, Structure Patterns and Behavior Patters (cf. [14] for informal description and [27] for the rCOS formalization) and refactoring rules (cf. [11] for informal discussion and [27,44] for the rCOS formalization). Some of these patterns introduce new classes and decompose classes. We plan to implement a library of design patterns and refactoring rules in the rCOS tool. A design pattern or a refactoring rule has conditions on the model before and after the transformation. The application of the corresponding automated transformation generates these conditions as proof obligations to be proved by using theorem proving and/or model checking. This is how verification tools are to be integrated into the rCOS tool, that is, we propose a tool suite for *verification which is integrated into model transformations*.

Effect of transformations. The application of these transformations to an analysis model refines the interface sequence diagram to an *object sequence diagram* and the conceptual class diagram to a *design class diagram* in which methods of a class are introduced, and the functionality specification of interface methods into invocations of the newly introduced methods of the classes and *specification statements* of these methods in their classes. Now the design class diagram can be automatically produced for a transformation, but the automatic generation of the object sequence diagram is harder and yet to be automated.

Discussion. Pre-processing of the functionality specification of the method is needed so that it is decomposed into specifications in terms of primitive responsibilities. This sounds unrealistic. However, the practical engineering guidance that the precondition of a method is mainly to check conditions on existing objects and the postcondition are mainly about which new objects were created, old object deleted, what attributes modification were made on which existing objects.

Our experience is that with the class diagram this guidance actually helps in writing and understanding the functionality specification of a method in a normal form that is essentially a *conjunction of disjunctions of sequential compositions of primitive responsibilities* [6]. An expression can represent a significant computation, such as the greatest common divisor of two integers or the shortest paths between two nodes of a graph, and it needs to be coded by a *programmer*. We have also defined refinement rules to transform universally and existentially quantified specification statements into loops [6]. These rules can be easily automated.

Component-based design. The designer takes the object-oriented design model and identifies "permanent objects" and decides if they should be made into components according to their features. The features include if they logically represents existing components, hardware devices, external subsystems, or they can be reused in different models of this application and other applications. A permanent object that aggregates a large amount objects and functionality is also suggested to be made into a component. The identification of objects as subcomponents in the object sequence diagram also defines the interfaces among the subcomponents. We can then *abstract* the object sequence diagram into a *component sequence diagram* by hiding the object interactions inside the identified components [43]. This step of abstraction is yet to be automated as a transformation. It generates the invariant that none of these objects is *null* for proof that the identified objects are indeed permanent. The execution of this transformation will also produce a *component diagram* representing the original component as the composition of the identified components. Reuse of existing designed components is also decided when applying the transformation. This generates proof obligations for checking fitness and composability of existing design components.

Platform specific design and implementation. The Designer decides on components that should maintain persistent data, and defines database mappings and primary keys for these components, and plans database queries to access the persistent data.

The model of component-based design obtained from the object-oriented design services as the *platform independent design* and employs direct object method interactions. The Designer studies the nature of the components, such as their distribution and deployment requirements, and decides the concrete interaction mechanisms and middlewares for individual interfaces.

5 Concluding Remarks

We have presented the motivation, the theme, the features and challenges of the rCOS theory and its tool support. The presentation is mostly informal, but what we are delighted about is that all the informal concepts, artifacts and design activities have their formalized versions in the rCOS theory and formulated in the roadmap of the design of the rCOS tool (cf. [17,3,6]). We take this as a promising sign of the research as we believe a theory and a tool can be effective only when they can be embedded into a practical software engineering processes. Except for application specific significant algorithms, nearly all the code can be automatically generated from a well specified design model. Also, transformations from a platform independent design to a platform specific design with existing industry standards can be mostly automated.

We have left out the discussion about *system integration* of the paper due the lack of space. System integration is mainly about the design of GUI objects and hardware controller that need to interact with each other and with the domain components. Modeling, analysis and design of these interactions can be done in a pure event-based modeling theory and its tools support for embedded information systems design [6].

There is a long way to fulfill our vision on the design and the implementation of the tool set out in [26]. The main challenge is still in the automation of model transformations from analysis models to platform independent design models. It is not enough to only provide a library of implementation transformations, but more importantly, the tool should provide guiding information on which rule is to be used. It is also difficult to support consistent and correct reuse of already designed methods when applying the Expert pattern to design a method, and the reuse of already designed components when designing a new component.

On the engineering side, our tool shows the same aspects and their respective problems as the software engineering discipline we would like to apply it to: development of the tool requires understanding of formal methods to correctly encode the requirements and algorithms, such as transformations and their preconditions, just as the model designers need to understand how to properly model a contract (including technicalities that might be necessary to make a problem actually amenable to the model checked), or write relational functionality specifications. While our progress in the tool development is steady but slow, we feel that it is well in scope of a commercial application from usability, presentation and documentation, included guided story-telling of use cases. As a matter of fact, we have *only* been able to make this progress with our limited resources because we have been harnessing existing infrastructure such as UML for modelling, and the QVT language for transformation, just as we want developers to harness existing theories in their designs and their validation.

Acknowledgements. We would like to thank our colleagues in the rCOS team (cf. `http://rcos.iist.unu.edu`) for their collaboration and discussions. We are in particular grateful to Anders P. Ravn for his suggestions and comments. We also thank the anonymous reviewers for their comments.

References

1. Booch, G.: Object-oriented analysis and design with applications. Addison-Wesley, Reading (1994)
2. Burstall, R., Goguen, J.: Putting theories together to make specifications. In: Reddy, R. (ed.) Proc. 5th Intl. Joint Conf. on Artificial Intelligence, pp. 1045–1058. Department of Computer Science, Carnegie-Mellon University, USA (1977)

3. Chen, X., He, J., Liu, Z., Zhan, N.: A model of component-based programming. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 191–206. Springer, Heidelberg (2007)
4. Chen, X., Liu, Z., Mencl, V.: Separation of concerns and consistent integration in requirements modelling. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, pp. 819–831. Springer, Heidelberg (2007)
5. Chen, Z., Hannousse, A.H., Hung, D.V., Knoll, I., Li, X., Liu, Y., Liu, Z., Nan, Q., Okika, J.C., Ravn, A.P., Stolz, V., Yang, L., Zhan, N.: Modelling with relational calculus of object and component systems–rCOS. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example. LNCS, vol. 5153, pp. 116–145. Springer, Heidelberg (2008)
6. Chen, Z., Liu, Z., Ravn, A.P., Stolz, V., Zhan, N.: Refinement and verification in component-based model driven design. Science of Computer Programming 74(4), 168–196 (2008); Special Issue on the Grand Challenge. UNU-IIST TR 388
7. Chen, Z., Liu, Z., Stolz, V.: The rCOS tool. In: Fitzgerald, J., Larsen, P.G., Sahara, S. (eds.) Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop, Newcastle University. CS-TR-1099 in Technical Report Series (May 2008)
8. Chen, Z., Morisset, C., Stolz, V.: Specification and validation of behavioural protocols in the rCOS modeler. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 387–401. Springer, Heidelberg (2010)
9. CWB. The concurrency workbench, `http://homepages.inf.ed.ac.uk/perdita/cwb/`
10. Dijkstra, E.: Notes on structured programming. In: Dahl, O.-J., Hoare, C.A.R., Dijkstra, E.W. (eds.) Structured Programming. Academic Press, London (1972)
11. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
12. Frederick, P., Brooks, J.: No silver bullet: essence and accidents of software engineering. Computer 20(4), 10–19 (1987)
13. Frederick, P., Brooks, J.: The mythical man-month: after 20 years. IEEE Software 12(5), 57–60 (1995)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
15. Goguen, J., Burstall, R.: Institutions: abstract model theory for specification and programming. Journal of ACM 39(1), 95–146 (1992)
16. Jifeng, H., Li, X., Liu, Z.: Component-based software engineering. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 70–95. Springer, Heidelberg (2005); UNU-IIST TR 330
17. He, J., Liu, Z., Li, X.: rCOS: A refinement calculus of object systems. Theor. Comput. Sci. 365(1-2), 109–142 (2006); UNU-IIST TR 322
18. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
19. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall, Englewood Cliffs (1998)
20. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)
21. Jones, C.B.: Systematic Software Development using VDM. Prentice-Hall, Englewood Cliffs (1990)
22. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems 16(3), 872–923 (1994)

23. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn. Prentice-Hall, Englewood Cliffs (2001)
24. Li, D., Li, X., Liu, J., Liu, Z.: Validation of requirements models by automatic prototyping. J. Innovations in Systems and Software Engineering 4(3), 241–248 (2008)
25. Liu, Z., Kang, E., Zhan, N.: Composition and refinement of components. In: Post event Proceedings of UTP 2008. LNCS. Springer, Heidelberg (to appear, 2009)
26. Liu, Z., Mencl, V., Ravn, A.P., Yang, L.: Harnessing theories for tool support. In: Proc. of the Second Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006), pp. 371–382. IEEE Computer Society, Los Alamitos (2006); Full version as UNU-IIST Technical Report 343
27. Long, Q., Qiu, Z., Liu, Z.: Formal use of design patterns and refactoring. In: Margaria, T., Steffen, B. (eds.) International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Communications in Computer and Information Science, vol. 17, pp. 323–338. Springer, Heidelberg (2008)
28. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems: specification. Springer, Heidelberg (1992)
29. Manna, Z., Pnueli, A.: The temporal logic of reactive systems: safety. Springer, Heidelberg (1992)
30. Milner, R.: Communication and concurrency. Prentice-Hall, Englewood Cliffs (1989)
31. Morgan, C.C.: Programming from Specifications. Prentice-Hall, Englewood Cliffs (1994)
32. Ossher, H., Tarr, P.: Using multidimensional separation of concerns to (re)shape evolving software. Commun. ACM 44(10), 43–50 (2001)
33. Paige, R., Brooke, P., Ostroff, J.: Metamodel-based model conformance and multiview consistency checking. ACM Trans.Softw.Eng.Methodol. 16(3), 11 (2007)
34. Parnas, D.: On the criteria to be used to decompose systems into modules. Communication of ACM 15, 1053–1058 (1972)
35. Peter, L.: The Peter Pyramid. William Morrow, New York (1986)
36. Roscoe, A.W.: Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1997)
37. Schneider, A.: The B-method. Masson (2001)
38. Sommerville, I.: Software Engineering, 6th edn. Addison-Wesley, Reading (2001)
39. SRI. PVS specification and verification system, `http://pvs.csl.sri.com/`
40. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, Reading (1997)
41. Wirsing, M., Banâtre, J.-P., Hölzl, M., Rauschmayer, A. (eds.): Software-Intensive Systems and New Computing Paradigms. LNCS, vol. 5380. Springer, Heidelberg (2008)
42. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall, Englewood Cliffs (1996)
43. Yang, L., Stolz, V.: Integrating refinement into software development tools. In: Pu, G., Stolz, V. (eds.) 1st Workshop on Harnessing Theories for Tool Support in Software. Electr. Notes in Theor. Comp. Sci., vol. 207, pp. 69–88. Elsevier, Amsterdam (2008); UNU-IIST TR 385
44. Zhao, L., Liu, X., Liu, Z., Qiu, Z.: Graph transformations for object-oriented refinement. Formal Aspects of Computing 21(1-2), 103–131 (2009)

Termination in Higher-Order Concurrent Calculi*

Romain Demangeon[1], Daniel Hirschkoff[1], and Davide Sangiorgi[2]

[1] ENS Lyon, Université de Lyon, CNRS, INRIA, France
[2] Università di Bologna, Italy

Abstract. We study termination of programs in concurrent higher-order languages. A higher-order concurrent calculus combines features of the λ-calculus and of the message-passing concurrent calculi. However, in contrast with the λ-calculus, a simply-typed discipline need not guarantee termination; and, in contrast with message-passing calculi such as the π-calculus, divergence can be obtained even without a recursion (or replication) construct.

We first consider a higher-order calculus where only processes can be communicated. We propose a type system for termination that borrows ideas from termination in Rewriting Systems (and following the approach to termination in the π-calculus in [DS06]). We then show how this type system can be adapted to accommodate higher-order functions in messages. Finally, we address termination in a richer calculus, that includes localities and a passivation construct, as well as name-passing communication. We illustrate the expressiveness of the type systems on a few examples.

1 Introduction

A system is terminating when it cannot perform an infinite number of transition steps. Termination is a difficult property to ensure: for instance, the termination of a rewriting system is not decidable in the general case. The problem of termination has been widely studied in sequential languages, including higher-order ones such as the λ-calculus, employing static analysis and especially type systems.

Ensuring termination for concurrent and mobile systems is much more challenging, as such systems are rarely confluent. The presence of mobility, under the form of an evolving topology of communication (new servers can be created, information travels across the system along dynamically evolving connections), adds even more complexity to the task. Previous works on this subject [YBH04, San06, DS06] rely on type systems to ensure termination in a concurrent context, in the setting of the π-calculus (π). In some of these systems, weights are assigned to π-calculus channels, and typability guarantees that, at

* This work has been supported by the European Project FET-GC II IST-2005-16004 Sensoria, and by the french ANR projects "CHoCo" and "Complice".

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 81–96, 2010.

each reduction step that involves the firing of a replicated term, the total weight associated to the process decreases.

In this work, we want to address the problem of termination in languages that include powerful primitives for distributed programming. The most important primitive that we focus on is *process passing*, that is, the ability to transmit an entity of computation along messages. We therefore study higher-order concurrent languages, and focus on the Higher-Order π-calculus, HOpi [San92], as working formalism to analyse termination in this setting.

To our knowledge, there exists no result on termination for higher-order concurrent processes. In some sense, formalisms like HOpi combine features from both the λ-calculus and the π-calculus, and ensuring termination in such a setting involves the control of difficulties related both to the higher-order aspects and to the concurrency aspects of the model.

In contrast with name-passing concurrent languages such as the π-calculus, where recursion (or a similar operator such as replication) is needed in order to have non-terminating programs, in HOpi, similarly to the λ-calculus, non-termination can show up already in the fragment without recursion. As an example, consider the following process:

$$Q_0 = P_0 \mid \overline{a}\langle P_0\rangle, \qquad \text{where} \qquad P_0 = a(X).(X \mid \overline{a}\langle X\rangle)$$

(P receives a process on channel a, spawns the received process and emits a copy of this process on a again). Q_0 can only reduce to itself, giving rise to a divergence.

Also, in contrast with the λ-calculus, where termination is ensured by adopting a simple type discipline, such as that of the simply-typed λ-calculus, which rules out recursive types, the HOpi process Q_0 is typable without resorting to recursive types (Q_0 is a process of simply-typed HOpi, where name a is used to carry processes, and the variables used are process variables).

To sum up, calculi like HOpi put together ideas from π-calculus and λ-calculus, and in both these calculi termination has been studied (using type systems). We cannot however directly adapt existing ideas. On the one hand, the type systems for termination in the π-calculus essentially impose constraints on the recursion (or replication) operators; we cannot directly adopt the idea in HOpi because HOpi has no recursion. On the other hand, the type systems for termination in the λ-calculus put constraints on self-applications, notably by forbidding recursive types. We cannot directly adopt these either, because of non-terminating examples like the one above. Indeed, there is no explicit self-application in Q_0, and Q_0 is actually typable in the simplest of the type systems of HOpi, which corresponds to the simply-typed discipline of the λ-calculus, without recursive types.

The goal of this paper is to propose more refined type disciplines, that allow us to rule out non-terminating programs such as the one above while retaining a non-trivial expressiveness.

A solution could be to exploit the standard encoding of HOpi in π [SW01], that respects termination, and use it, together with existing type systems for π, to infer termination in HOpi. However this would not be applicable in extensions of HOpi that are not encodable in π (or that appear difficult to encode), for instance, in distributed versions of the calculus. If one wishes to handle models for distributed computing (including explicit locations and mobility of locations), the techniques and type systems for termination should be directly formulated on HOpi. Further, a direct formulation would allow one to make enhancements of the techniques that are tailored to (and therefore more effective on) higher-order concurrency. We nevertheless analyse the approach via the encoding in the π-calculus in Sect. 2.3, to compare it with our system in terms of expressiveness.

In this paper, we first (Sect. 2) analyse termination in HOpi$_2$, a higher-order calculus where processes are the only values exchanged. We propose a type system for termination using techniques from term-rewriting, in which termination is guaranteed by a decreasing weight associated to processes. This is also the approach followed in [DS06] for termination in the π-calculus. The technical details and the proofs are however rather different, for the reasons outlined earlier (e.g., name-passing vs process passing, absence of replication or recursion). We present the basic type system, make some assessment of its expressiveness, and describe a few important refinements (though only briefly, due to lack of space).

The system for HOpi$_2$ is a starting point, from which we build a similar type system for HOpi$_\omega$, a richer higher-order calculus where the values communicated also include higher-order functions (Sect. 3 – the names HOpi$_2$ and HOpi$_\omega$ are inspired from [SW01]). The additional constructs for functions have to be controlled in order to rule out diverging behaviours.

These results pave the way for the study of a further and much richer extension, the calculus we call PaPi (Sect. 4). PaPi is equipped with powerful primitives that are found in formalisms for global computing: in addition to standard name-passing (as in the π-calculus) and higher-order communication, we also handle explicit localities and *passivation*. Passivation is the operation of intrusively capturing a running computation, in order to be able to modify the process being executed (for instance to discard, duplicate or update it). We provide several examples to illustrate the expressive power given by the combination of primitives in PaPi. Analysing and controlling interaction in PaPi is a challenging task. We discuss how the ideas we developed to control process passing in HOpi$_2$ and HOpi$_\omega$ can be combined with the approach to name passing of [DS06] in order to guarantee termination.

2 HOpi$_2$

This section is dedicated to the study of HOpi$_2$, a basic higher-order process calculus, with processes as the only communication values.

2.1 The Calculus

We shall use symbols P, Q, R, S for processes, X, Y for process variables, and names a, b, c for channels.

$$\frac{}{\overline{a}\langle Q\rangle.P_1 \mid a(X).P_2 \to P_1 \mid P_2[Q/X]}$$

$$\frac{P_1 \to P_1'}{P_1 \mid P_2 \to P_1' \mid P_2} \qquad \frac{P \to P'}{(\nu c)P \to (\nu c)P'} \qquad \frac{Q \equiv P \qquad P \to P' \qquad P' \equiv Q'}{Q \to Q'}$$

Fig. 1. The Operational Semantics of HOpi$_2$

The grammar for processes of HOpi$_2$ is the following:

$$P ::= \mathbf{0} \mid P|P \mid \overline{a}\langle P\rangle.P \mid a(X).P \mid X \mid (\nu c)P \ .$$

Structural congruence ($\equiv$) is defined in the standard way on HOpi$_2$. We shall omit trailing occurrences of $\mathbf{0}$ in processes of the form $\overline{a}\langle P\rangle.\mathbf{0}$. Reduction is defined by the rules of Fig. 1. $Q[P/X]$ stands for the capture avoiding substitution of variable X with process P in Q. A process P is *terminating* if there exists no infinite sequence of reductions emanating from P. We suppose that all processes we shall manipulate obey a *Barendregt convention*: all bound names are pairwise distinct and different from all free names (similar notations and conventions will be adopted for the calculi we study in the next sections).

To see how HOpi$_2$ processes interact, consider $S_1 = \overline{a}\langle\overline{b}\langle\mathbf{0}\rangle.\mathbf{0}\rangle.\overline{a}\langle b(Z).\mathbf{0}\rangle$ and $S_2 = a(X).a(Y).(X \mid Y)$. S_1 is a process which sends on a the code of a process emitting $\mathbf{0}$ on b, and then sends on a the code of a process receiving on b. S_2 is a process which upon reception of two processes on a (in sequence) executes these in parallel. Process $S_1 \mid S_2$ performs two reductions to become $\overline{b}\langle\mathbf{0}\rangle.\mathbf{0} \mid b(Z).\mathbf{0}$, after which a synchronisation on b can take place.

As discussed above, recursive outputs ("self-emissions") can lead to diverging behaviours in HOpi$_2$: in process Q_0 from Sect. 1, a process containing an output on a is sent over channel a itself in $\overline{a}\langle P_0\rangle$. Our type system, in Sect. 2.2, puts constraints on self-emissions in order to control divergence.

2.2 A Type System to Ensure Termination in HOpi$_2$

The types for channels are of the form $Ch^n(\diamond)$, where $\diamond$ is interpreted as the type of processes (throughout the paper, we use the syntax $Ch(T)$ to denote the type of a channel carrying values of type T), and n is a natural number, called the *level* of the channel being typed. We use Γ to range over typing contexts, that are lists of typing hypotheses. If $a : Ch^n(\diamond)$ belongs to Γ, we write $\Gamma(a) = Ch^n(\diamond)$, and $lvl_\Gamma(a) = n$. Processes (and process variables in Γ) are typed using simply a natural number.

Figure 2 presents the rules of our type system for HOpi$_2$. (This system, and all systems we shall study in the paper, are *syntax directed*: there is one typing rule per syntactic construct. We shall exploit this when referring to the typing rules by only mentioning the construct they deal with.) The actual control takes place in the output rule, where we ensure that the level of the transmitted process

$$\frac{\Gamma(X)=n}{\Gamma \vdash X : n} \qquad \frac{\Gamma, c : Ch^{k}(\diamond) \vdash P : n}{\Gamma \vdash (\nu c)P : n} \qquad \frac{\Gamma \vdash P_1 : n_1 \qquad \Gamma \vdash P_2 : n_2}{\Gamma \vdash P_1 \mid P_2 : \max(n_1, n_2)}$$

$$\frac{}{\Gamma \vdash \mathbf{0} : 0} \qquad \frac{\begin{array}{c}\Gamma \vdash P : k \qquad \Gamma \vdash Q : m \\ lvl_{\Gamma}(a) = n \qquad k < n\end{array}}{\Gamma \vdash \overline{a}\langle P\rangle.Q : \max(m, n)} \qquad \frac{\begin{array}{c}\Gamma, X : k-1 \vdash P : n \\ lvl_{\Gamma}(a) = k\end{array}}{\Gamma \vdash a(X).P : n}$$

Fig. 2. HOpi$_2$: Typing Rules

is strictly smaller than the level of the carrying channel: this way, we exclude "self-emissions". This discipline is at the basis of the termination proof: when a communication is performed, an output of weight n is traded for possibly several new outputs appearing in the process, that all have a smaller weight.

We can check that process Q_0 from Sect. 1 is ruled out by our system: as P_0 contains an output on a, its level is at least the level of a. As a consequence, the output rule forbids P_0 to be sent on a itself, and Q_0 is not typable.

To establish soundness of our type system, we introduce a measure on typing derivations that decreases along reductions. We use notation $\mathcal{D} : (\Gamma \vdash P : n)$ to mean that $\mathcal{D}$ is a derivation of the typing judgment $\Gamma \vdash P : n$. Below and in the remainder of the paper, $\uplus$ will stand for multiset union.

Definition 1. *If $\mathcal{D} : (\Gamma \vdash P : n)$, we define $m(\mathcal{D})$ by induction over the structure of $\mathcal{D}$ as follows (to ease presentation, we take advantage of the fact that the type system is syntax-directed, and reason according to the shape of P):*

- $m_{\mathcal{D}}(\mathbf{0}) = m_{\mathcal{D}}(X) = \emptyset$*;*
- $m_{\mathcal{D}}(P_1 \mid P_2) = m_{\mathcal{D}}(P_1) \uplus m_{\mathcal{D}}(P_2)$
- $m_{\mathcal{D}}((\nu c)P) = m_{\mathcal{D}}(a(X).P) = m_{\mathcal{D}}(P)$*;*
- $m_{\mathcal{D}}(\overline{a}\langle P\rangle.Q) = m_{\mathcal{D}}(Q) \uplus \{n\}$ *if $lvl(a) = n$ according to Γ.*

We note that if $\mathcal{D} : (\Gamma \vdash P : n)$ and $P \equiv P'$, there exists $\mathcal{D}'$ s.t. $\mathcal{D}' : (\Gamma \vdash P' : n)$ and $m(\mathcal{D}) = m(\mathcal{D}')$. $<_{mul}$ denotes the multiset extension of the standard ordering on integers (e.g., $\{2, 2\} <_{mul} \{3\}$).

Lemma 1. *If $\mathcal{D} : (\Gamma \vdash P : n)$, then $m(\mathcal{D}) <_{mul} \{n+1\}$.*

Proposition 1. *If $\mathcal{D} : (\Gamma \vdash P : n)$ and $P \rightarrow P'$ then there exist $\mathcal{D}'$ and $n' \leq n$ such that $\mathcal{D}' : (\Gamma \vdash P : n')$ and $m(\mathcal{D}') <_{mul} m(\mathcal{D})$.*

Proof (Sketch). We reason by induction on the derivation of the transition of P. The most interesting case is when $P = \overline{a}\langle P_1\rangle.Q_1 \mid a(X).Q_2 \rightarrow P' = Q_1 \mid Q_2[P_1/X]$. By the typing hypothesis, we get $lvl(a) = n$ and $\mathcal{D}_0 : (\Gamma \vdash P_1 : k)$ for some $\mathcal{D}_0$ and k. We can build a typing derivation $\mathcal{D}'$ for P' such that there exists c satisfying $m(\mathcal{D}') = m(\mathcal{D}) \setminus \{n\} \uplus m(\mathcal{D}_0)^c$, where $m(\mathcal{D}_0)^c$ stands for the multiset union of c copies of $m(\mathcal{D}_0)$. Indeed, an output on a in P is erased along the reduction, and there are possibly several copies of P_1 which appear in P': c

is defined as the number of occurrences of X in Q_2 which do not appear inside messages. From typability of $\overline{a}\langle P_1\rangle.Q_1$, we get $k < n$, and from Lemma 1 we deduce $m(\mathcal{D}_0) <_{mul} \{n\}$. Thus $m(\mathcal{D}') <_{mul} m(\mathcal{D})$. □

Corollary 1. *If $\Gamma \vdash P : n$, then P terminates.*

Proof. Follows from Proposition 1 and the fact that the multiset extension of a terminating order is terminating. □

2.3 An Analysis of the Type System for HOpi$_2$

Typing via Encoding into $\boldsymbol{\pi}$. We now compare the expressiveness of our type system with the expressiveness induced on HOpi$_2$ by the translation into π and the existing type system [DS06] for the π-calculus.

Translating HOpi$_2$ processes. We use (an adaption of) the standard encoding of HOpi$_2$ into the π-calculus [San92] (see also [Tho96]). The target calculus of our encoding is the simply typed monadic π-calculus, with `unit` as base type (the unique value of type `unit` is noted ★), and where replication is allowed only on inputs. The encoding is rather standard – we only recall the clauses for input, output and process variables (an unambiguous correspondence between HOpi$_2$ process variables and their counterpart as π names is implicitly assumed):

$$[\![a(X).P]\!] = a(x).[\![P]\!] \qquad [\![X]\!] = \overline{x} \qquad [\![\overline{a}\langle Q\rangle.P]\!] = (\nu h_a)\ \overline{a}\langle h_a\rangle.(\ [\![P]\!] \mid\ !h_a.[\![Q]\!]\ )$$

A higher-order output action $\overline{a}\langle Q\rangle.P$ is translated into the emission of a new name (h_a), which intuitively is the address where process Q can be accessed. Interactions on h_a and x, noted using CCS prefixes, actually involve the transmission of the unique value of type `unit`.

Proposition 2. *For any HOpi$_2$ process P, P terminates iff $[\![P]\!]$ terminates.*

Typing the encoding. We rely on the first type system of [DS06] to type the encoding of a HOpi$_2$ process. This type system assigns levels to names, in order to control replicated processes. If we call $os(P)$ the multiset consisting of channel names that are used as subject of an output in a π process P, and where the output does not occur under a replication, then $!a(x).P$ is well-typed if the level of a is strictly greater than the level of all names in $os(P)$. We write $\Gamma \vdash_{\mathrm{pi}} P$ for typability in the π-calculus according to [DS06]. All processes typable using this type system are terminating.

There exist HOpi$_2$ processes that can be proved to terminate using the type system for HOpi$_2$, but whose encoding fails to be typable using the type system for π. A very simple example is given by $R_0 = a(X).\overline{a}\langle X\rangle$. We indeed have

$$[\![R_0]\!] \quad = \quad a(h_X).(\nu h_a)\,\overline{a}\langle h_a\rangle.!h_a.\overline{h_X}\ ,$$

which is not typable: indeed, h_X and h_a necessarily have the same type (both are transmitted on a), which prevents subprocess $!h_a.\overline{h_X}$ from being typable.

This example suggests a way to establish a relationship between the type systems in HOpi$_2$ and in π. Consider for that the type system for HOpi$_2$ obtained by replacing rule (**In**) in Figure 2 with the following one, the other rules remaining unchanged (the typing judgment for this modified type system shall be written $\Gamma \vdash_{\mathbf{m}} P : n$):

$$(\mathbf{In}')\ \frac{\Gamma, X : k \vdash_{\mathbf{m}} P : n \qquad lvl_\Gamma(a) = k}{\Gamma \vdash_{\mathbf{m}} a(X).P : n}$$

Clearly, the modified type system is more restrictive, that is, $\Gamma \vdash_{\mathbf{m}} P : n$ implies $\Gamma \vdash P : n$, but not the converse (cf. process R_0 seen above).

Using this system, we can establish the following property, that allows us to draw a comparison between typability in HOpi$_2$ and in the π-calculus:

Proposition 3. *Let S be a HOpi$_2$ process. If $\Gamma \vdash_{\mathbf{m}} S : m$, then there exists Δ, a typing context for π, such that $\Delta \vdash_{\mathrm{pi}} [\![S]\!]$.*

Proof (Sketch). The crux of this proof is the correspondence between the typing of output actions in HOpi$_2$ and the typing of replications in π.

The encoding seen above induces a translation of HOpi$_2$ typing contexts as follows (type checking the encoding of a restricted term induces similar typing assumptions):

$$[\![\emptyset]\!] = \emptyset \qquad\qquad [\![\Gamma, X : n]\!] = [\![\Gamma]\!], x : Ch^n(\mathtt{unit})$$

$$[\![\Gamma, a : Ch^n(\diamond)]\!] = [\![\Gamma]\!], a : Ch^n(Ch^n(\mathtt{unit}))$$

To show that $\Gamma \vdash_{\mathbf{m}} S : m$ implies $[\![\Gamma]\!] \vdash_{\mathrm{pi}} [\![S]\!]$, we focus on replicated terms in $[\![S]\!]$. Every replication appearing in $[\![S]\!]$ corresponds to the encoding of an output of S. It therefore appears in a context of the form $(\nu h_a)\ \overline{a}\langle h_a\rangle.(!h_a.[\![P]\!] \mid [\![Q]\!])$, corresponding to an output action $\overline{a}\langle P\rangle.Q$ occurring in S. As $\Gamma \vdash_{\mathbf{m}} S : m$, the rule for output gives $\Gamma' \vdash_{\mathbf{m}} a : Ch^n(\diamond)$ and $\Gamma' \vdash_{\mathbf{m}} P : m'$ for some Γ', with $m' < n$. This means that $[\![\Gamma']\!] \vdash_{\mathrm{pi}} h_a : Ch^n(\mathtt{unit})$ and $\forall b \in os([\![P]\!]), [\![\Gamma']\!](b) = Ch^m(T)$ with $m \leq m'$. Thus $n > m$, and the replicated input at h_a is well-typed (in π). □

Remark 1 (The limits of our type system)

Symmetrically to the example process R_0 seen above, there exist terms that can be typed via the encoding, but that are rejected by our type system: consider

$$R_1 = \overline{a}\langle\overline{a}\langle\mathbf{0}\rangle\rangle \mid a(X).\mathbf{0} \quad\text{and}\quad R_2 = a(X).b(Y).X \mid \overline{a}\langle\overline{a}\langle\mathbf{0}\rangle\rangle \mid \overline{b}\langle\mathbf{0}\rangle\ .$$

None of these processes is typable, because they contain "self-emissions" (an output action on channel a occurring inside a process emitted on a). However, R_1 and R_2 are terminating. Their encodings in π are

$$[\![R_1]\!] = (\nu h_a)\ \overline{a}\langle h_a\rangle.!h_a.(\nu h'_a)\ \overline{a}\langle h'_a\rangle.!h'_a.\mathbf{0} \mid a(x).\mathbf{0} \qquad\text{and}$$
$$[\![R_2]\!] = a(x).b(y).\overline{x} \mid (\nu h_a)\ \overline{a}\langle h_a\rangle.!h_a.(\nu h'_a)\ \overline{a}\langle h'_a\rangle.!h'_a.\mathbf{0} \mid (\nu h_b)\ \overline{b}\langle h_b\rangle.!h_b.\mathbf{0}\ ,$$

which are both typable using the system of [DS06]. A suitable assignment for R_1 is, e.g., $lvl(a) = 1$, $lvl(h_a) = lvl(h'_a) = 2$; both replications are typed as the first one trades a name of level 2 for a name of level 1 and the second one has no output at all in its continuation. R_2 can be typed with the same level assignment, and $lvl(h_b) = lvl(h'_b) = 2$.

It thus appears that self-emissions can be innocuous, while they are systematically rejected by the system of Sect. 2. Self-emissions in R_1 and R_2 are reminiscent of recursive calls in continuations of replicated π processes, like, e.g., in $!a(x).b(y).\overline{a}\langle y\rangle$. It turns out that constructions like the one we find in R_2 show up in examples (see Remark 2 below).

As pointed out in the Introduction, a direct type system can be the basis for refinements and extensions. Indeed, both the refinements discussed at the end of this section, and the extensions presented in Sect. 4 allow us to handle processes that go well beyond those that can be treated via encodings into the pi-calculus.

Towards More Expressive Type Systems. We now discuss some possible refinements of our type system that allow us to overcome the limitations we have presented. Some of these refinements are inspired by the type systems developed in [DS06], some are specific to our higher-order setting.

A first enrichment consists in attaching two pieces of information to a channel, instead of simply a level. First, a channel a has a *weight*, which stands for the contribution of active outputs on a to the global weight of a process. For instance, in the process $P_1 = \overline{a_1}\langle P_2\rangle$, with $P_2 = \overline{b_1}\langle Q_1\rangle \mid \overline{b_2}\langle Q_2\rangle$, the global weight of P_2 is equal to the sum of the weights attached to names b_1 and b_2. Second, a channel a has a *capacity*, which is an upper bound on the weight of processes that may be sent on a: P_1 is well-typed provided the capacity of a_1 is greater than the weight of P_2. This approach can be related to the observations we have made above about $[\![R_1]\!]$ and $[\![R_2]\!]$, where the level of a (resp. h_a) plays the rôle of the weight (resp. the capacity).

As a second enrichment, we use multisets of natural numbers to represent the weight and the capacity attached to a channel, as well as the type attached to a process. The rules defining a type system that includes these two enrichments are presented on Fig. 3, where M, N denote multisets of natural numbers. In the rule for output, M_2 plays the rôle of the capacity (which must dominate the weight of Q), and M_1, the weight of the output on a, is combined with the weight (M) of the continuation process P. As an example, if the outputs on a (resp. on b) weight $\{1\}$ (resp. $\{2\}$), the process $\overline{a}\langle P\rangle \mid \overline{a}\langle P'\rangle \mid \overline{b}\langle Q\rangle$ has type $\{2, 1, 1\}$.

In the rule for input, $o(M, P, X)$ stands for the multiset obtained by computing the multiset union of as many copies of M as there are occurrences of X that do not appear in a message in P. Formally:

$$\begin{array}{l}
o(M, \mathbf{0}, X) = \emptyset \\
o(M, X, X) = M, \qquad o(M, Y, X) = \emptyset,\ Y \neq X \\
o(M, P_1 \mid P_2, X) = o(M, P_1, X) \uplus o(M, P_2, X) \\
o(M, a(Y).P, X) = o(M, P, X),\ Y \neq X, \qquad o(M, a(X).P, X) = \emptyset \\
o(M, \overline{a}\langle Q\rangle.P, X) = o(M, (\nu c)\ P, X) = o(M, P, X)
\end{array}$$

$$\frac{}{\Gamma \vdash \mathbf{0} : \emptyset} \qquad \frac{\Gamma(X) = M}{\Gamma \vdash X : M} \qquad \frac{\Gamma, c : Ch^{M_1,M_2}(\diamond) \vdash P : N}{\Gamma \vdash (\nu c)P : N}$$

$$\frac{\begin{array}{c}\Gamma \vdash P_1 : M_1 \\ \Gamma \vdash P_2 : M_2\end{array}}{\Gamma \vdash P_1 \mid P_2 : M_1 \uplus M_2} \qquad \frac{\begin{array}{cc}\Gamma \vdash P : M & \Gamma \vdash Q : N \\ \Gamma(a) = Ch^{M_1,M_2}(\diamond) & N <_{mul} M_2\end{array}}{\Gamma \vdash \overline{a}\langle Q\rangle.P : M \uplus M_1}$$

$$\frac{\Gamma, X : M_2 \vdash P : M \qquad \Gamma(a) = Ch^{M_1,M_2}(\diamond) \qquad o(M_2, P, X) <_{mul} M_1}{\Gamma \vdash a(X).P : M}$$

Fig. 3. Typing Processes using Multisets

Computing $o(M, P, X)$ is necessary because communication of a process Q can have the effect of spawning several copies of Q. Accordingly, the weight associated to the channel transmitting Q must be strictly greater than the total weight of the processes spawned along consumption of the message.

To establish soundness of this type system, we rely as previously on a measure on terms. The measure of a process P is given by the multiset union of the weights associated to all names that are used in output, for occurrences that are not themselves within a message in P. We show that the type of P (i.e., the multiset given by the typing judgment for P) is always greater than the measure of P. Intuitively, this is the case because the process variables contribute to the type, but not to the measure. In a reduction of the form $\overline{a}.\langle Q\rangle.P_1 \mid a(X).P_2 \rightarrow P_1 \mid P_2[Q/X]$, with $\Gamma(a) = Ch^{M_1,M_2}(\diamond)$ and $\Gamma \vdash Q : N$, an output of type M_1 is consumed and a process Q of type N is spawned in P_2 for each occurrence of X in P_2. The typing rule for outputs enforces $M_2 >_{mul} N$ and the typing rule for input enforces $M_1 >_{mul} o(M_2, P_2, X)$. This entails that M_1 is greater than the multiset union of the measure of each process Q spawned in P_2. Thus the measure globally decreases, which guarantees termination.

We can check that using this type system, certain forms of "self-emission", like in $R_1 = \overline{a}\langle\overline{a}\langle\mathbf{0}\rangle\rangle \mid a(X).\mathbf{0}$, can be typed. If we assign weight $\{1\}$ and capacity $\{2\}$ to a, the output is well-typed because process $\overline{a}\langle\mathbf{0}\rangle$ has weight $\{1\}<_{mul} \{2\}$. The input is also well-typed as the weight of a is greater than $o(\{2\}, \mathbf{0}, X) = \emptyset$.

We have studied a third refinement of our type system, defined for a higher-order formalism with a primitive construct for replication. This in principle does not add expressiveness to the calculus, because replication is encodable in HOpi$_2$ (using a process similar to Q_0 from Sect. 1). However, in terms of typability, having a primitive replication, and a dedicated typing rule for it, helps in dealing with examples. The type system to handle replication in presence of higher-order communications controls divergences that can arise both from self-emissions and from recursion in replications (as they appear in the setting of [DS06]). More details about this analysis are given in an extended version of this paper [DHS09].

A further refinement: handling successive input prefixes. Inspired by the third type system of [DS06], we can treat sequences of input prefixes as a kind of 'single input action', that has the effect of decreasing the weight of the process being executed.

Let us sketch the main idea behind this approach. Consider a process of the form $a_1(X_1).\dots.a_k(X_k).P$. We make sure that the weight associated to the sequence of inputs is strictly greater than the weight of the processes that are spawned in the continuation P. If $\Gamma(a_i) = Ch^{M_1^i, M_2^i}(\diamond)$, then the former quantity is equal to $M_1^1 \uplus \cdots \uplus M_1^k$. To compute the latter quantity, one has, like above, to take into account the multiplicity of the X_is (whose weight is given by typing) in P — again, we only consider occurrences of these process variables that are not within messages.

According to this approach, the overall weight of a process can temporarily increase along communications, before a sequence of inputs is consumed. However, each consumption of a whole sequence of inputs induces a global weight loss, thus ensuring termination (it can be shown that for a divergence to exist, there must be infintely many consumptions of whole sequences of inputs). Technically, the shape of the soundness proof follows the lines of the justification of a corresponding type system for the name-passing paradigm in [DHS08].

This way, process $R_2 = a(X).b(Y).X \mid \overline{a}\langle\overline{a}\langle\mathbf{0}\rangle\rangle \mid \overline{b}\langle\mathbf{0}\rangle$ can be typed by assigning type $Ch^{\{1\},\{1,1\}}(\diamond)$ to a and $Ch^{\{2\},\{2\}}(\diamond)$ to b. The input sequence is typed as the total weight of the sequence is $\{2,1\}$, and the total weight of the processes spawned is $\{1,1\}$ (one occurrence of X, no occurrence of Y). The outer output on a is typed as the weight of the object process is $\{1\}$ (an output on a) and the capacity of a is $\{1,1\}$. The inner output on a and the output on b are well-typed as the capacities of these two names are greater than $\emptyset$, the weight of $\mathbf{0}$.

Remark 2 (Encoding the choice operator). To illustrate the expressiveness of the resulting type system, we show in [DHS09] the typability of Nestmann and Pierce's protocol for modelling (separate) choice [NP00] — precisely, the protocol adapted to a higher-order calculus. From the termination viewpoint, this protocol is interesting because it is non-trivial, and because its termination is not a straightforward property to establish, as the protocol involves some forms of backtracking. Also, when rewritten in the higher-order paradigm (more precisely, in an extension of HOpi$_\omega$– see Sect. 3), the protocol makes use of some patterns or combinations of operators that are delicate for termination (in particular, a pattern similar to $a(X).b(Y).X$ in the above example).

3 HOPi$_\omega$: Transmitting Higher-Order Functions

The Calculus. We now present HOpi$_\omega$, a calculus inspired from HOPi$^{\mathtt{unit},\rightarrow,\diamond}$ in [SW01]. The main difference between HOpi$_\omega$ and HOpi$_2$ is that the values communicated in HOpi$_\omega$ can be ★, the unique element of type `unit`, or functions (precisely parametrised processes) of arbitrarily high order (the order indicating the level of arrow nesting in the type). The grammar for processes and values (we use metavariables v, w, not to be confused with names, to range over values, and x, y to range over variables) is the following:

$$P := \mathbf{0} \mid P|P \mid \overline{a}\langle v\rangle.P \mid v\lfloor v\rfloor \mid a(x).P \mid (\nu a)P \qquad\qquad v = \bigstar \mid x \mapsto P$$

$x \mapsto P$ is a function from values to processes, and $v\lfloor w\rfloor$ is the application of a function to its argument. We will restrict ourselves to meaningful usages of (higher-order) functions, which can be ensured by adopting a standard type discipline (see below).

The operational semantics of HOpi$_\omega$ is given by the rules below (rules for closure w.r.t. parallel composition, restriction, and structural congruence are omitted): communication involves the transmission of a value, and β-reduction takes place when a function is applied to a value.

$$\frac{}{\overline{a}\langle v\rangle.Q_1 \mid a(x).Q_2 \to Q_1 \mid Q_2[v/x]} \qquad\qquad \frac{}{(x \mapsto P)\lfloor v\rfloor \to P[v/x]}$$

HOpi$_2$ processes can be seen as HOpi$_\omega$ processes by replacing communication of processes with communication of values of type $\mathtt{unit} \to \diamond$, and, accordingly, usages of process variables with an application to $\bigstar$. For instance, the diverging example Q_0 in HOpi$_2$ becomes $\overline{a}\langle x \mapsto P\rangle \mid P$ where $P = a(y).(y\lfloor\bigstar\rfloor \mid \overline{a}\langle y\rangle)$.

The following is an example HOpi$_\omega$ process:

$$\begin{aligned} P \;=\; & \overline{a}\langle x \mapsto (x\lfloor\bigstar\rfloor \mid x\lfloor\bigstar\rfloor)\rangle \;\mid\; \overline{b_1}\langle x_1 \mapsto \overline{c}\langle\bigstar\rangle\rangle.\overline{b_2}\langle x_2 \mapsto c(z).\mathbf{0}\rangle \\ \mid\; & b_1(y_1).b_2(y_2).a(y_3).(y_3\lfloor y_1\rfloor \mid y_3\lfloor y_2\rfloor)\ . \end{aligned}$$

Channel c has type $Ch(\mathtt{unit})$, channels b_1, b_2 have type $Ch(\mathtt{unit} \to \diamond)$ (see the grammar for types below), and channel a has type $Ch((\mathtt{unit} \to \diamond) \to \diamond)$. P can do two communications on b_1 and b_2. Then, a function (in this case, a duplicator) can be transmitted on a, and successively applied to the functions sent on b_1 and b_2 (corresponding to processes emitting and receiving on c). After these three reductions, we obtain $\overline{c}\langle\bigstar\rangle \mid \overline{c}\langle\bigstar\rangle \mid c(z).\mathbf{0} \mid c(z').\mathbf{0}$, which can still do two synchronisations.

Type System. The grammar for types for HOpi$_\omega$ includes types for values, given by $T ::= \mathtt{unit} \mid (T \to^n \diamond)$, and channel types, of the form $Ch^n(T)$. We restrict ourselves to using only *well-formed* value types, defined as follows:

Definition 2 (Well-formed value types). *We say that T is a* well-formed value type *at level n w.r.t. a typing context Γ (written $Lvl_\Gamma(T) = n$ or simply $Lvl(T) = n$ when there is no ambiguity on Γ), whenever either $T = \mathtt{unit}$ and $n = 0$, or T' is a well-formed value type at level n', $T = T' \to^n \diamond$ and $n' < n$.*

The rules defining our type system for HOpi$_\omega$ are presented in Fig. 4. As in Sect. 2, types are annotated with a level, and the type assigned to a process is given by a natural number. The type of a process P is bound to dominate both the maximum level of outputs contained in P (not occurring inside a message), and the maximum level associated to function v_1, in applications $v_1\lfloor v_2\rfloor$ that occur in P not inside a message.

Typing values

$$\frac{}{\Gamma \vdash \bigstar : \mathtt{unit}} \qquad \frac{\Gamma, x : T \vdash P : n}{\Gamma \vdash x \mapsto P : T \to^{n+1} \diamond}$$

Typing processes

$$\frac{}{\Gamma \vdash \mathbf{0} : 0} \qquad \frac{\Gamma, a : Ch^{k}(T) \vdash P : n}{\Gamma \vdash (\nu a)P : n} \qquad \frac{\Gamma \vdash P_1 : n_1 \qquad \Gamma \vdash P_2 : n_2}{\Gamma \vdash P_1 \mid P_2 : \max(n_1, n_2)}$$

$$\frac{\begin{array}{c}\Gamma \vdash v_1 : T \to^{n} \diamond \\ \Gamma \vdash v_2 : T\end{array}}{\Gamma \vdash v_1 \lfloor v_2 \rfloor : n} \qquad \frac{\begin{array}{c}\Gamma, x : T \vdash P : n \\ \Gamma(a) = Ch^{k}(T)\end{array}}{\Gamma \vdash a(x).P : n} \qquad \frac{\begin{array}{c}\Gamma \vdash v : T \qquad \Gamma \vdash P : n' \\ \Gamma(a) = Ch^{n}(T) \qquad Lvl(T) = k \qquad n > k\end{array}}{\Gamma \vdash \overline{a}\langle v \rangle.P : \max(n, n')}$$

Fig. 4. Typing Rules for HOpi$_\omega$

Soundness. As before, we associate to a process a measure that decreases along reductions. Relying as above on $os(P)$, the multiset of names used in output subject position in P, does not work, because β-reduction may let $os(P)$ grow.

Definition 3 (Measure on processes in HOpi$_\omega$). *Let P be a well-typed HOpi$_\omega$ process. We define $\mathcal{M}(P) = os(P) \uplus fun(P)$, where: (i) $os(P)$ is the multiset of the levels of the channel names that are used in an output in P, without this output occurring in message position. (ii) $fun(P)$ is defined as the multiset union of all $\{k\}$, for all $v_1 \lfloor v_2 \rfloor$ occurring in P not within a message, such that v_1 is of type $T \to^{k} \diamond$.*

Proposition 4 (Soundness). *If $\Gamma \vdash P : n$ for some HOpi$_\omega$ process P, then P terminates.*

Proposition 4 is established by observing that $\mathcal{M}(P)$ decreases at each step of transition:

- If the transition is a communication, the continuations of the processes involved in the communication contribute to the global measure the same way they did before communication, because a type preserving substitution is applied. $\mathcal{M}(P)$ decreases because an output has been consumed.
- If the transition is a β-reduction involving a function of level k, a process of level strictly smaller than k is spawned in P. Therefore, all new messages and active function applications that contribute to the measure are of a level strictly smaller than l, and $\mathcal{M}(P)$ decreases.

4 Controlling Communication and Passivation

PaPi: A Calculus with Locations and Passivation. The objective of this section is to study termination in presence of further constructs that are known to be challenging in the semantics of higher-order concurrent languages, notably constructs of locations (i.e., explicit spatial distribution) and of passivation. We consider a calculus, which we refer to as PaPi (for '*Pa*ssivation *Pi*-calculus'), that integrates such constructs with the higher-order features of HOpi$_2$ and the name-passing capabilities of the π-calculus.

In PaPi, names belong to two sorts: they are either channels or locations. We use a, b, c to denote channels, and l to denote locations. We let n stand for any name, be it used as a channel or as a location, and names x, y, z will denote name variables. The syntax of PaPi is as follows:

$$\begin{array}{rcl} P & ::= & \mathbf{0} \mid P|P \mid \overline{a}\langle n\rangle.P \mid a(x).P \mid !a(x).P \\ & & \mid l[P] \mid l(X) \triangleright P \mid \overline{a}\langle P\rangle.P \mid a(X).P \mid (\nu n)P \ . \end{array}$$

Note that replication is allowed only on name-passing input prefixes. $l[P]$ stands for the process P running at location l (locations can be nested). The construct $l(X) \triangleright P$ corresponds to passivation, that is, the operation that consists in capturing a computation running at location l, calling it X, and proceeding according to P. Passivation can be found in calculi like Kells [SS05, HHH$^+$08] or Homer [HGB04].

The operational semantics of PaPi is described by the following reduction rules (we omit the rules for closure of reduction w.r.t. structural congruence, restriction and parallel composition):

$$\frac{}{\overline{a}\langle n\rangle.P \mid a(x).Q \to P \mid Q[n/x]} \qquad \frac{}{\overline{a}\langle P\rangle.Q_1 \mid a(X).Q_2 \to Q_1 \mid Q_2[P/X]}$$

$$\frac{}{l[Q] \mid l(X) \triangleright P \to P[Q/X]} \qquad \frac{P \to P'}{l[P] \to l[P']}$$

It has to be noted that we do not claim here that the combination of primitives provided in PaPi (first and higher-order message passing, localised interaction, passivation) makes this calculus a proposal for a model for distributed or component based programming. Indeed, important interaction mechanisms such as communication between distant locations, or subjective mobility, are not available in PaPi.

Our primary goal is instead to study how the constructs of PaPi, which have the advantage of being presented in a rather simple way, can be taken into account in our termination analysis. We believe that the way we handle these can be smoothly adapted to small variations: for instance, typing distant communication in kπ [HHH$^+$08] should be done pretty much like we type local communication in PaPi.

We now provide a few examples of PaPi processes to illustrate typical idioms that can be programmed using passivation.

$$\mathit{Dup} \quad c(r).l(X) \triangleright \big(\, l[X] \mid (\nu l')\,(\overline{r}\langle l'\rangle \mid l'[X])\,\big)$$

$$\mathit{Res} \quad c(l).l(X) \triangleright l[P_0] \qquad\qquad \mathit{DynUpd} \quad c(l).d(X).(\, l(Y) \triangleright l[X]\,)$$

$$\mathit{Coloc} \quad l_1(X) \triangleright \big(l_2(Y) \triangleright (l_1[X|Y] \mid l_2[\mathbf{0}])\big)$$

Dup performs code duplication: when a message is received on channel c, the computation running at location l is duplicated, and the location of the new copy is sent back on r, the channel transmitted along c.

Process *Res* (reset): upon reception of a location name l along c, the computation taking place at l is replaced with P_0, that can be considered as a start state. Essentially the same "program" can be used when we want to replace the code running at l with a new version, that is transmitted along some channel d: this is a form of dynamic update (process *DynUpd*).

"Co-localisation": processes running at locations l_1 and l_2 are put together, and computation proceeds within location l_1. This might trigger new interactions between formerly separated processes. This is a form of *objective mobility* (running computations are being moved around).

Termination in PaPi. In PaPi, divergences arise both from recursion in usages of the passivation and process-passing mechanisms, and from recursive calls in the continuation of replicated (name-passing) inputs. We control the latter source of divergences by resorting to the type discipline of [DS06], while the former is controlled by associating levels to locations and to process-carrying channels.

However, the mere superposition of these two systems (of Sect. 2.2 and of [DS06]) does not work, as the two mechanisms can cooperate to produce divergences. Indeed, consider the non terminating process $P_1 = l(X) \triangleright !a.X \mid l[\overline{a}] \mid \overline{a}$ (channel a, which is used in a CCS-like fashion, carries values of type `unit`). The usages of passivation (which can be treated as a form of process passing) and name passing in P_1 are unfortunately compliant with the principles of the aforementioned type systems. In this particular case, we must take into account the fact that X can be instantiated by a process containing an output on a channel having the same level as a. More generally, we must understand how the two type systems can interact, in order to avoid diverging behaviours.

In PaPi, every entity (process, location, name-passing channel and process-passing channel) is given a level which is used to control both sources of divergences. The base types for names are `unit` and **`loc`** (for location names). The level of a name-passing channel a corresponds to the maximum level allowed for the continuation P in a replicated input of the form $!a(x).P$. The level of a process-passing channel corresponds to the maximum level of a process sent on this channel. The level of a location corresponds to the maximum level a process executing at this location can have. In turn, the level of a process P corresponds to the maximum level of messages and locations that occur in P neither within a higher-order output nor under a replication.

The rules defining the type system for termination in PaPi are given in Fig. 5. As far as typing termination is concerned, higher-order inputs (resp. outputs) are typed like passivations (resp. located processes). We can moreover remark that this type system subsumes the type system of [DS06] for the π-calculus: if a π process P is typable according to [DS06], then it is typable as a PaPi process.

Remark 3. It has to be noted that the type system we present can be made more expressive by exploiting ideas from Sect. 2.3. Indeed, what we control here

$$\frac{}{\Gamma \vdash \mathbf{0} : 0} \qquad \frac{\Gamma(X) = m}{\Gamma \vdash X : m} \qquad \frac{\Gamma, v : T \vdash P : m}{\Gamma \vdash (\nu v)\ P : m}$$

$$\frac{\Gamma \vdash P_1 : m_1 \qquad \Gamma \vdash P_2 : m_2}{\Gamma \vdash P_1 \mid P_2 : \max(m_1, m_2)} \qquad \frac{\Gamma, X : k-1 \vdash P : m \qquad \Gamma(l) = \mathbf{loc}^k}{\Gamma \vdash l(X) \triangleright P : m}$$

$$\frac{\Gamma(l) = \mathbf{loc}^k \qquad \Gamma \vdash Q : m' \qquad k > m'}{\Gamma \vdash l[Q] : k} \qquad \frac{\Gamma, X : k-1 \vdash P : m \qquad \Gamma(a) = Ch^k(\diamond)}{\Gamma \vdash a(X).P : m}$$

$$\frac{\begin{array}{c}\Gamma \vdash P : m \qquad \Gamma \vdash Q : m' \\ \Gamma(a) = Ch^k(\diamond) \qquad k > m'\end{array}}{\Gamma \vdash \overline{a}\langle Q\rangle.P : \max(k, m)} \qquad \frac{\Gamma, x : T \vdash P : m \qquad \Gamma(a) = Ch^k(T)}{\Gamma \vdash a(x).P : m}$$

$$\frac{\begin{array}{c}\Gamma \vdash P : m \qquad \Gamma \vdash v : T \\ \Gamma(a) = Ch^k(T)\end{array}}{\Gamma \vdash \overline{a}\langle v\rangle.P : \max(k, m)} \qquad \frac{\begin{array}{c}\Gamma, x : T \vdash P : m \\ \Gamma(a) = Ch^k(T) \qquad k > m\end{array}}{\Gamma \vdash !a(x).P : 0}$$

Fig. 5. Typing Rules for PaPi

using a unique level for names could be refined by associating channels with three natural numbers: one is its weight, and the other two are interpreted as capacities, to control the two sources of recursion: the weight of name passing outputs on one side, and the weight of process passing outputs and located processes on the other side. In what we have presented, these three components of the type of a name are merged into a single one.

Termination. For lack of space, we do not present the soundness proof of our type system for PaPi. It essentially follows the same strategy as in the previous sections. At its core is the definition of a measure on processes, that takes into account the contribution of locations and first- and higher-order messages that do not occur within a message. We then show that this measure decreases along reductions, which finally gives:

Proposition 5. *If $\Gamma \vdash P : m$ for a PaPi process P, then P terminates.*

Examples of typing. Process P_1 seen above cannot be typed. The typing rule for locations forces the level of the location l to be strictly greater than $lvl(a)$ when typing $l[\overline{a}]$. The typing rule of passivation forces the level of l to be equal to $1 + lvl(X)$. Thus $lvl(X) \leq lvl(a)$ and the typing rule for replicated inputs cannot be applied to $!a.X$.

For process *Coloc* to be typable, $lvl(l_1)$, the level assigned to l_1, should be greater than $lvl(l_2)$. In this case, we can observe that it is safe to take two processes running in separate locations and let them run in parallel, as *Coloc* does: while this might trigger new interactions (inter-locations communication is forbidden in PaPi), this is of no harm for termination.

5 Concluding Remarks

In this paper, we have analysed termination in higher-order concurrent languages, using the higher-order π-calculus as a core formalism to build the basis of our type systems. For future work, we plan to examine how the type systems we have presented can be adapted to existing process calculi in which processes can be exchanged in communications or can move among locations such as, e.g., Ambients [CG98], Homer [HGB04], Kells [SS05, HHH$^+$08]. Another question we would like to address is type inference; for this, [DHKS07] could serve as a starting point.

References

[CG98] Cardelli, L., Gordon, A.D.: Mobile Ambients. In: Nivat, M. (ed.) FOSSACS 1998. LNCS, vol. 1378, pp. 140–155. Springer, Heidelberg (1998)

[DHKS07] Demangeon, R., Hirschkoff, D., Kobayashi, N., Sangiorgi, D.: On the Complexity of Termination Inference for Processes. In: Barthe, G., Fournet, C. (eds.) TGC 2007 and FODO 2008. LNCS, vol. 4912, pp. 140–155. Springer, Heidelberg (2008)

[DHS08] Demangeon, R., Hirschkoff, D., Sangiorgi, D.: Static and Dynamic Typing for the Termination of Mobile Processes. In: Proc. of IFIP TCS 2008. Springer, Heidelberg (2008)

[DHS09] Demangeon, R., Hirschkoff, D., Sangiorgi, D.: Termination in Higher- Order Concurrent Calculi (2009) (long version of this paper) (in preparation)

[DS06] Deng, Y., Sangiorgi, D.: Ensuring Termination by Typability. Information and Computation 204(7), 1045–1082 (2006)

[HGB04] Hildebrandt, T., Godskesen, J.C., Bundgaard, M.: Bisimulation Congruences for Homer — a Calculus of Higher Order Mobile Embedded Resources. Technical Report TR-2004-52, Univ. of Copenhagen (2004)

[HHH$^+$08] Hirschkoff, D., Hirschowitz, T., Hym, S., Pardon, A., Pous, D.: Encapsulation and Dynamic Modularity in the Pi-calculus. In: Proc. of the PLACES 2008 workshop. ENTCS. Elsevier, Amsterdam (2008) (to appear)

[NP00] Nestmann, U., Pierce, B.C.: Decoding Choice Encodings. Information and Computation 163(1), 1–59 (2000)

[San92] D. Sangiorgi. Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms. PhD Thesis, University of Edinburgh, 1992.

[San06] Sangiorgi, D.: Termination of Processes. Mathematical Structures in Computer Science 16(1), 1–39 (2006)

[SS05] Schmitt, A., Stefani, J.-B.: The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 146–178. Springer, Heidelberg (2005)

[SW01] Sangiorgi, D., Walker, D.: The π-calculus: a Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)

[Tho96] Thomsen, B.: Calculi for Higher Order Communication Systems. PhD Thesis, University of London (1996)

[YBH04] Yoshida, N., Berger, M., Honda, K.: Strong Normalisation in the Pi-Calculus. Information and Computation 191(2), 145–202 (2004)

Typing Asymmetric Client-Server Interaction*

Franco Barbanera[1], Sara Capecchi[2], and Ugo de'Liguoro[2]

[1] Dipartimento di Matematica e Informatica, Università di Catania
barba@dmi.unict.it
[2] Dipartimento di Informatica, Università di Torino
{capecchi,deligu}@di.unito.it

Abstract. We investigate client-server interaction where duties and rights of the parties are asymmetric, in the sense that the client is allowed to abort any session before the server has completed, but not vice versa. This implies that the client can interact with any server offering at least what she is looking for, but possibly more.

We formalize such asymmetry in the setting of session types via a form of subtyping in depth, which we call prefix relation. This is apparently conflicting with the rigid duality imposed by session types; nonetheless the resulting system retains all basic correctness properties.

Moreover, the system we propose highlights interesting aspects concerning the flow of communication inside a session. In particular it reveals that usual subtyping theories cannot be extended by means of prefix, which turns out to be a different concept.

Keywords: Process calculi, Type Systems, Session Types , Client/ Server Interaction Protocols, Subtyping, Contro/Covariance.

1 Introduction

Client/server is the relationship between two software applications in which one of them, the client, addresses its request of a service to the other one, the server, which is expected to fulfill the request. It is an intrinsically asymmetric relationship, not just because there is one interacting end that provides a service and the other that makes use of it. There are also differences in the rights and duties of the parties: it is indeed unreasonable to prohibit to the client to abort the connection at any time, while it would be unfair to admit such a behavior on the server side.

In [4] a theory of *contracts* is proposed in order to formalize the search for a service on the web and to discipline the client/server relationship. A contract is an abstract specification of the service. A client will *comply* with a service if it will successfully terminate any interaction with the service, which however might provide more. Since (the dual of) a contract is able to specify the interaction protocol also on the client side, the compliance of the client with the server can be checked by formally proving that the dual of the client contract is (with some

[*] This work has been partially supported by the MIUR project EOS-DUE.

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 97–112, 2010.

simplification) the initial part of the server contract. The same asymmetry is at the heart of the subcontract relationship studied in [4].

Following the suggestion in [15], we consider types as a natural candidate to formalize contracts, and more precisely the session types introduced in [8], a type system for a dialect of the π-calculus adding primitives to handle sessions. A session is an abstraction of a sequence of communications through a private channel between two parties. It is created by connecting over a session channel (often called *live* channel) in such a way that both privacy and duality are guaranteed. The type system is then used to abstract a discipline of the interaction into a *session type* and to ensure safe handshaking-communications.

The idea is that of changing the interpretation of the typing $x : S$, where S is a session type, according to the cases of x being the server or the client end in a session. In the server case S represents its duties, namely the commitment to an interaction which is at least of the shape (and the length) represented by S. If x is instead the client end, then S represents the client's rights, telling that it is entitled to ask at most an interaction of that shape.

According to the system in [8] sessions have to be symmetric. A simple example of symmetric interaction is between the following sketchy calculator server and one of its possible clients:

$$
\begin{aligned}
\mathit{CalcServer}_1 =_{\mathrm{def}} \mathsf{accept}\ a(x).x \triangleright \{\ & \mathsf{add} : x?(n).x?(m).x![n+m],\\
& \dots\\
& \mathsf{div} : x?(n).x?(m).x![n\ \mathsf{div}\ m]\}
\end{aligned}
$$

$$\mathit{CalcClient}_1 =_{\mathrm{def}} \mathsf{request}\ a(x).x \triangleleft \mathsf{div}.x![21].x![5].x?(n)$$

The type of the name x along which a client might interact with $\mathit{CalcServer}_1$ is

$$S_{\mathrm{server1}} = \&\langle \mathsf{add} : ?(\mathsf{int})\, ?(\mathsf{int})\, ![\mathsf{int}]\mathsf{end}, \dots, \mathsf{div} : ?(\mathsf{int})\, ?(\mathsf{int})\, ![\mathsf{int}]\mathsf{end}\rangle,$$

while the type of the x on the side of $\mathit{CalcClient}_1$ is exactly its dual:

$$S_{\mathrm{client1}} = \overline{S_{\mathrm{server1}}} = \oplus\langle \mathsf{add} : ![\mathsf{int}]\, ![\mathsf{int}]\, ?(\mathsf{int})\mathsf{end}, \dots, \mathsf{div} : ![\mathsf{int}]\, ![\mathsf{int}]\, ?(\mathsf{int})\mathsf{end}\rangle$$

However, the service request of $\mathit{CalcClient}_1$ would be satisfied also by the server

$$
\begin{aligned}
\mathit{CalcServer}_2 =_{\mathrm{def}} \mathsf{accept}\ a(x).x \triangleright \{\ & \mathsf{add} : x?(n).x?(m).x![n+m],\\
& \dots\\
& \mathsf{div} : x?(n).x?(m).x![n\ \mathsf{div}\ m].x![n\ \mathsf{mod}\ m]\}
\end{aligned}
$$

whose typing of x is just "longer" than S_{server1}:

$$S_{\mathrm{server2}} = \&\langle \mathsf{add} : ?(\mathsf{int})\, ?(\mathsf{int})\, ![\mathsf{int}]\mathsf{end}, \dots, \mathsf{div} : ?(\mathsf{int})\, ?(\mathsf{int})\, ![\mathsf{int}]\, ![\mathsf{int}]\mathsf{end}\rangle$$

Can the concept of "being longer" be caught by means of subtyping? It is indeed tempting to try to extend the subtyping theory of session types introduced in [7] so that S_{client1} is a subtype of S_{client2}, that we can assume for x because of narrowing rule. However, as we shall argue in the sequel, this is not the case, and in fact the idea of extending a protocol represented by a type by some sort of subtyping in depth (whereas the subtyping of [7] is essentially in width w.r.t. branching/selection types) reveals to be of a different nature. Because of this we introduce a new relation among session types which we call *prefix*, axiomatize it and study its effect w.r.t. the "more liberal" system of session types in [20]. We choose the latter system as the basis of the present study also because it uses polarized channel names as private channels inside a session (like in [7] before). We profit of polarities to mark the server and the client end of a session, which we type differently w.r.t. the prefix relation.

The resulting system, when restricted to first-order sessions (i.e. transmitting just values and labels for selection among a branching of processes), satisfies a property we call *weak compliance*, which roughly says that a typed server cannot exhaust its actions on a channel before the dually typed client does. We call it weak compliance in contrast to the strong compliance considered in [4] because, as it is well known, session types do not guarantee deadlock freeness, so that we cannot expect that the client will also complete the interaction on its side. Technically the result is achieved by means of a simple semantics of session types interpreting types as sets of usages of channel names.

When considering higher order sessions (exchanging channel names of other sessions), the problem of variance of the types for input/output of a session is faced. We throughly study the possibility of adapting to the prefix relation the same variance rules that do hold in the case of subtyping in [7]. From the problems that arise we distill our solution, consisting in having both input and output types covariant w.r.t. the prefix, but we constrain the rules used to type the transmission of the (private) channel of a session. The resulting system nicely formalizes delegation, which in the world of object oriented programming is the act of transparently passing the interaction with a third party from one object providing a first part of a service, to another object that provides its continuation.

The full system of asymmetric session types retains all the basic properties of the original, symmetric system, namely subject reduction and error freeness. Unfortunately the semantics of first order types does not extend to higher order sessions, and we are able to prove a slightly weaker result of compliance in that case.

2 The π_S-Calculus and Its Operational Semantics

Session types are a type system for a dialect of the π-calculus introduced in [8]. From now on we will call π_S-calculus the π-calculus extended with primitives structuring sessions. π_S-calculus names are divided into sorts: *names* $a, b, ..,$ $x, y \ldots \in \mathcal{N}$, (*polarized*) *channels* $\kappa^p, \kappa'^p, \ldots \in \mathcal{C}$, *labels* $l, l', \ldots \in \mathcal{L}$. The term syntax is presented in Figure 1, where we omit definitions (and hence recursion) in order to focus on the essential issues and to keep the technical development simpler. Such a syntax is based on the second system introduced in [20], where polarized names are used for (private) channels, building over an idea that first appeared in [6] (a preliminary version of [7].) Notice that we take into account

$e ::=$	**ground exprs**	$p, q ::= + \mid -$	**polarities**
$\mid x \mid \mathsf{true} \mid \mathsf{false} \mid 0 \mid 1 \mid \ldots$	vars and consts	$u, u' ::= x \mid \kappa$	
$\mid \mathsf{not}(e) \mid e + e' \mid \ldots$	operators	$k, k' ::= a \mid x \mid \kappa \mid \kappa^p$	**names, channels**
$\pi ::=$	**prefixes**	$P, Q ::=$	**processes**
$\mid \mathsf{request}\ a(x)$	session request	$\mid \mathbf{0}$	inaction
$\mid \mathsf{accept}\ a(x)$	session acceptance	$\mid \pi.P$	prefixed process
$\mid k![e]$	data sending	$\mid k \triangleright \{l_1 : P_1, .., l_n : P_n\}$	label branching
$\mid k?(x)$	data reception	$\mid P \vert Q$	parallel composition
$\mid \mathsf{throw}\ k[k']$	channel sending	$\mid (\nu\kappa)P$	restriction
$\mid \mathsf{catch}\ k(x)$	channel reception	$\mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional branch
$\mid k \triangleleft l$	label selection		

Fig. 1. Term syntax of the π_S-calculus (without recursion)

the monadic version of the system, since our notions and results simply extend to the polyadic case.

Communications belonging to a session are opened through a public name and pursued through a private channel, specific to the session, which is created at connection time. The syntax used for the initiation of a session is:

$$\mathsf{request}\ a(x).P \ \mid\ \mathsf{accept}\ a(x).Q \tag{1}$$

A process opening a session by a request action over a will be called a *client* w.r.t. the name a; symmetrically, a *server* w.r.t. a opens the session using accept. Of course a process can be client and server w.r.t. different names, and even w.r.t. the same name that could be used to open several sessions.

A *polarized channel name* is a channel name decorated with a polarity: κ^p, $\kappa^{\overline{p}}$ where $p \in \{+, -\}$ and $\overline{+} = -$ and $\overline{-} = +$. We use polarities not only, as in [7,20], to couple the owners of a private channel, but also to distinguish between the client's and server's end of it ('$-$' for the client and '$+$' for the server). Indeed the process term in (1) reduces to:

$$(\nu\kappa)(\{\kappa^-/x\}P \mid \{\kappa^+/x\}Q) \tag{2}$$

where the channel name κ is fresh. Note that occurrences of channels are always polarized (even in the scope of a binder). The only use of unpolarized channels is in the binder operator $(\nu\kappa)$, which binds all the channels κ^p in its scope, regardless of p.

Free and bound names and channel names in P are denoted, respectively, by $\text{FN}(P)$ and $\text{BN}(P)$ and defined as in [20]. Substitution and α-congruence, written $P \equiv_\alpha Q$, are defined as usual.

The operational semantics of the calculus is usually defined in terms of a reduction relation up to structural congruence (see in particular [20], §§ 2.1 and 3.1). However, in order to make proofs simpler, we consider an equivalent LTS semantics. The labels of the LTS are from **Act**, the set of *actions* α, defined by:

$$\alpha ::= a(\kappa^+) \mid a[\kappa^-] \mid \kappa^p(v) \mid \kappa^p[v] \mid \kappa^p \triangleleft l \mid \kappa^p \triangleright l \mid \kappa\nu k \mid \tau.$$

where v is ambiguous for c, κ^q, κ, and k is either a name or a channel, possibly polarized; τ represents an internal action as for CCS. The dual action $\overline{\alpha}$ is defined only in the following cases:

$$\overline{\kappa^p(v)} = \kappa^{\overline{p}}[v] \qquad \overline{\kappa^p[v]} = \kappa^{\overline{p}}(v) \qquad \overline{\kappa^p \lhd l} = \kappa^{\overline{p}} \rhd l \qquad \overline{\kappa^p \rhd l} = \kappa^{\overline{p}} \lhd l$$

The relation $P \xrightarrow{\alpha} Q$ is then defined in Figure 2. It is an adaptation of the early semantics of the π-calculus (see e.g. [16]), where the symmetric of E-Par, E-Close and E-Link are omitted.

By denoting with $P \longrightarrow Q$ the *reduction semantics*, as defined in [20]§3.1, the proof of the following can be provided by essentially mimicking that of the Harmony Lemma in [19].

Proposition 1. *The LTS semantics and the reduction semantics are equivalent:*

1. *if* $P \equiv\xrightarrow{\alpha} P'$ *then* $P \xrightarrow{\alpha}\equiv P'$*;*
2. $P \longrightarrow P'$ *if and only if* $P \xrightarrow{\tau}\equiv P'$*.*

$$\dfrac{\kappa^+, \kappa^- \notin \textsc{fn}(P)}{\mathsf{accept}\ a(x)P \xrightarrow{a(\kappa^+)} \{\kappa^+/x\}P}\ \textsc{E-Acc} \qquad \dfrac{\kappa^+, \kappa^- \notin \textsc{fn}(P)}{\mathsf{request}\ a(x)P \xrightarrow{a[\kappa^-]} \{\kappa^-/x\}P}\ \textsc{E-Req}$$

$$\dfrac{}{\kappa^p?(x).P \xrightarrow{\kappa^p(c)} \{c/x\}P}\ \textsc{E-Input} \qquad \dfrac{e \downarrow c}{\kappa^p![e].P \xrightarrow{\kappa^p[c]} P}\ \textsc{E-Output}$$

$$\dfrac{l_i \in \{l_1, \ldots, l_n\}}{\kappa^p \rhd \{l_1 : P_1, .., l_n : P_n\} \xrightarrow{\kappa^p \rhd l_i} P_i}\ \textsc{E-Brn} \qquad \dfrac{}{\kappa^p \lhd l.P \xrightarrow{\kappa^p \lhd l} P}\ \textsc{E-Sel}$$

$$\dfrac{P \xrightarrow{\alpha} P' \quad \kappa \notin \alpha}{(\nu\kappa)P \xrightarrow{\alpha} (\nu\kappa)P'}\ \textsc{E-Res} \qquad \dfrac{P \xrightarrow{\kappa_1^p[\kappa_2^q]} P' \quad \kappa_1 \neq \kappa_2}{(\nu\kappa_2^q)P \xrightarrow{\kappa_1^p \nu \kappa_2^q} P'}\ \textsc{E-Open}$$

$$\dfrac{P \xrightarrow{\alpha} P' \quad \textsc{bn}(\alpha) \cap \textsc{fn}(Q) = \emptyset}{P|Q \xrightarrow{\alpha} P'|Q}\ \textsc{E-Par} \qquad \dfrac{P \xrightarrow{\kappa^p(\kappa')} P' \quad Q \xrightarrow{\kappa^{\overline{p}} \nu \kappa'} Q'}{P|Q \xrightarrow{\tau} (\nu\kappa')(P'|Q')}\ \textsc{E-Close}$$

$$\dfrac{e \downarrow \mathsf{true}}{\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} P}\ \textsc{E-IfT} \qquad \dfrac{e \downarrow \mathsf{false}}{\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} Q}\ \textsc{E-IfF}$$

$$\dfrac{P \xrightarrow{a(\kappa^+)} P' \quad Q \xrightarrow{a[\kappa^-]} Q'}{P|Q \xrightarrow{\tau} (\nu\kappa)(P|Q)}\ \textsc{E-Link} \qquad \dfrac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\overline{\alpha}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}\ \textsc{E-Com}$$

$$\dfrac{}{\mathsf{catch}\ \kappa_1^p(x).P \xrightarrow{\kappa_1^p(\kappa_2^q)} \{\kappa_2^q/x\}P}\ \textsc{E-Cat} \qquad \dfrac{}{\mathsf{throw}\ \kappa_1^p[\kappa_2^q].P \xrightarrow{\kappa_1^p[\kappa_2^q]} P}\ \textsc{E-Thr}$$

Fig. 2. Early LTS-operational semantics of π_S

$$\overline{S} = S \text{ if } S = \mathsf{end} \qquad \overline{\uparrow[S]} = \uparrow[S] \qquad \overline{?(T)S} = ![T]\overline{S} \qquad \overline{![T]S} = ?(T)\overline{S}$$

$$\overline{\&\langle l_1 : S_1, .., l_n : S_n\rangle} = \oplus\langle l_1 : \overline{S_1}, .., l_n : \overline{S_n}\rangle \qquad \overline{\oplus\langle l_1 : S_1, .., l_n : S_n\rangle} = \&\langle l_1 : \overline{S_1}, .., l_n : \overline{S_n}\rangle$$

Fig. 3. Dual session types

3 Asymmetric Session Types

Definition 1 (Types). *The sets of* types $\mathcal{T}$*, and of* session types $\mathcal{ST}$*, are defined according to the following grammar*

$$\begin{array}{rl} \textit{Type} & T ::= \mathsf{bool} \mid \mathsf{nat} \mid \mathsf{int} \mid \mathsf{real} \mid \ S \ \mid \uparrow[S] \\ \textit{Session type} & S ::= \ ?(T)S \mid ![T]S \mid \&\langle l_1 : S_1, \ldots, l_n : S_n\rangle \\ & \qquad \mid \oplus\langle l_1 : S_1, \ldots, l_n : S_n\rangle \mid \mathsf{end} \end{array}$$

A session type is *first order* if neither $?(T)S$ nor $![T]S$ occurs in it for any $T \in \mathcal{ST}$; it is *higher order* otherwise. We recall that the ordering of labels in $\&\langle l_1 : S_1, \ldots, l_n : S_n\rangle$ and $\oplus\langle l_1 : S_1, \ldots, l_n : S_n\rangle$ is immaterial. The operation $\overline{S}$ over $\mathcal{ST}$ is defined as in Figure 3. It is easy to check that $\overline{\overline{S}} = S$ for any $S \in \mathcal{ST}$.

Types are assigned to names according to the rules in Figure 4. Judgments have either the form $\Gamma \vdash x : T$, $\Gamma \vdash e : T$ or $\Gamma \vdash P \triangleright \Delta$, where $x \in \mathcal{N}$, e is an expression, P is a process and where Γ and Δ are typing contexts, i.e. finite mappings from names to types. In particular, $\Gamma(x) = T$ where $dom(\Gamma) \subseteq \mathcal{N}$, and $\Delta(\kappa^p) = S$ where $dom(\Delta) \subseteq \mathcal{C}$. As in [8], Δ is called a *typing*. Note that κ^+ and κ^- are considered as different, so that both may be in $dom(\Delta)$ for some typing Δ. Differently from [20]§3, the domain of a typing contains only polarized channels. A typing Δ is *completed* if $\Delta(\kappa^p) = \mathsf{end}$, for all $\kappa^p \in dom(\Delta)$; *balanced* if $\Delta(\kappa^p) = \overline{\Delta(\kappa^{\overline{p}})}$ whenever $\kappa^p, \kappa^{\overline{p}} \in dom(\Delta)$; *strictly balanced* if it is balanced and $\kappa^p \in dom(\Delta)$ implies $\kappa^{\overline{p}} \in dom(\Delta)$. In case $\kappa^p \notin dom(\Delta)$, $\Delta \cdot \kappa^p : S$ is defined as the typing Δ' such that $\Delta'(\kappa^p) = S$ and $\Delta'(\kappa'^q) = \Delta(\kappa'^q)$ if $\kappa'^q \neq \kappa^p$. $\Delta \cdot \Delta'$ denotes the component wise extension of the dot operation, which is hence defined only if $dom(\Delta) \cap dom(\Delta') = \emptyset$.

Beside the restriction of typings Δ to polarized channel names, the main difference w.r.t. the system in §3 of [20] concerns rules T-Acc and T-Req. In our system the bound name x in the conclusion is substituted for the polarized channel name κ^p appearing in the process $\{\kappa^p/x\}P$ of the premises. The polarity p is '+' in T-Acc and '−' in T-Req. Such requirements on polarities force to use the channel represented by x as a server's or client's end of a session, respectively.

Session types enforce a perfect symmetry of the server and client actions via rules T-Acc, T-Req and T-NewS: the first two, together with the additivity of rule T-Par w.r.t. the left hand typing Γ, ensure that a connection opened through a name a such that $a : \uparrow[S] \in \Gamma$ will use a channel κ^+ of type S on the

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \textsc{T-Name} \qquad \frac{\Delta \textit{ completed}}{\Gamma \vdash \mathbf{0} \triangleright \Delta}\ \textsc{T-Inact} \qquad \frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta \cdot \Delta'}\ \textsc{T-Par}$$

$$\frac{\Gamma \vdash a : \uparrow[S] \quad \Gamma \vdash \{\kappa^+/x\}P \triangleright \Delta \cdot \kappa^+ : S}{\Gamma \vdash \mathsf{accept}\ a(x).P \triangleright \Delta}\ \textsc{T-Acc}$$

$$\frac{\Gamma \vdash a : \uparrow[S] \quad \Gamma \vdash \{\kappa^-/x\}P \triangleright \Delta \cdot \kappa^- : \overline{S}}{\Gamma \vdash \mathsf{request}\ a(x).P \triangleright \Delta}\ \textsc{T-Req}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot \kappa^p : S \quad \Gamma \vdash e : T}{\Gamma \vdash \kappa^p![e].P \triangleright \Delta \cdot \kappa^p : ![T]S}\ \textsc{T-Send} \quad \frac{\Gamma \cdot x : T \vdash P \triangleright \Delta \cdot \kappa^p : S}{\Gamma \vdash \kappa^p?(x).P \triangleright \Delta \cdot \kappa^p : ?(T)S}\ \textsc{T-Rcv}$$

$$\frac{\{\Gamma \vdash P_i \triangleright \Delta \cdot \kappa^p : S_i\}_{i=1,\dots,n}}{\Gamma \vdash \kappa^p \triangleright \{l_1 : P_1, \dots, l_n : P_n\} \triangleright \Delta \cdot \kappa^p : \&\langle l_1 : S_1, \dots, l_n : S_n\rangle}\ \textsc{T-Br}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot \kappa^p : S_i \quad i \in \{1, \dots n\}}{\Gamma \vdash \kappa^p \triangleleft l_i.P \triangleright \Delta \cdot \kappa^p : \oplus\langle l_1 : S_1, \dots, l_n : S_n\rangle}\ \textsc{T-Sel}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot \kappa^+ : S \cdot \kappa^- : \overline{S}}{\Gamma \vdash (\nu\kappa)P \triangleright \Delta}\ \textsc{T-NewS} \quad \frac{\Gamma \vdash P \triangleright \Delta \quad \kappa \notin \textit{dom}(\Delta)}{\Gamma \vdash (\nu\kappa)P \triangleright \Delta}\ \textsc{T-NewS}'$$

Fig. 4. The First-Order Type System

server side, and a channel κ^- of type $\overline{S}$ on the client side. Since this channel is private and a process like (1) reduces to one like (2), the same correspondence is required to introduce the restriction $(\nu\kappa)$ in rule T-NewS.

We wish now to formalize the bias toward the client by breaking such a symmetry, allowing sessions in which the client might do less than the server actually offers. A first attempt could consist in using subtyping (we write $T <: T'$), as introduced in [18] for I/O types and studied in [7] specifically for session types.

For the sake of the subsequent discussion we recall that, w.r.t. the subtyping of I/O types, of which it saves the covariance of the input and contravariance of the output types, the theory in [7] includes:

$$\frac{S_i <: S_i' \ \ (\forall i \leq n) \quad n \leq m}{\&\langle l_1 : S_1, .., l_n : S_n\rangle <: \&\langle l_1 : S_1', .., l_n : S_m'\rangle} \qquad \frac{S_i <: S_i' \ \ (\forall i \leq n) \quad n \leq m}{\oplus\langle l_1 : S_1, .., l_n : S_m\rangle <: \oplus\langle l_1 : S_1', .., l_n : S_n'\rangle}$$

making the & covariant and the $\oplus$ contravariant in width respectively (though both are covariant in depth).

Since we wish to embody the idea that a server might be ready to do more than it is declared by its "interface" S, we should add the axiom $\mathsf{end} <: S$ to the theory in [7] and consider a restriction of the narrowing rule (the dual of the subsumption rule customary in the typed λ-calculus: see e.g. [17]) to positive channel names only:

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot \kappa^{+} : S \quad S' <: S}{\Gamma \vdash P \triangleright \Delta \cdot \kappa^{+} : S'}\ \textsf{T-Narrow}$$

Now, being & the dual of $\oplus$ and being $?(\_)\_$ the dual of $![\_]\_$ (with $?(\_)\_$ covariant and $![\_]\_$ contravariant in the first argument), it is the case that if $S <: S'$ then $\overline{S'} <: \overline{S}$. However, this key property of subtyping relative to duality is incompatible with the axiom $\textsf{end} <: S$.

Theorem 1. *There is no consistent theory of subtyping, extending the theory in [7], which includes the axiom* $\textsf{end} <: S$ *and satisfies the principle that if* $S <: S'$ *then* $\overline{S'} <: \overline{S}$.

Proof. Toward a contradiction assume that if $S <: S'$ then $\overline{S'} <: \overline{S}$ holds; since $\textsf{end} <: \overline{S}$ is an instance of the axiom $\textsf{end} <: S$, for any S we have that $S = \overline{\overline{S}} <: \overline{\textsf{end}} = \textsf{end}$. Therefore, by transitivity of $<:$, it is $S <: S'$ for any $S, S' \in \mathcal{ST}$. □

Theorem 1 leads us to introduce a new relation among session types: $S \preccurlyeq S'$, that we call *prefix relation*. Roughly, if $S \preccurlyeq S'$ then any interaction pattern typed by S is the initial part of a pattern typed by S'.

Definition 2 (Prefix Relation over First Order Session Types). *The* prefix *relation over first-order session types,* $S \preccurlyeq S'$ *(read "S is a prefix of S'") is defined as the least preorder satisfying the following axiom and rules*

$$\frac{}{\textsf{end} \preccurlyeq S} \qquad \frac{S \preccurlyeq S'}{?(T)S \preccurlyeq ?(T)S'} \qquad \frac{S \preccurlyeq S'}{![T]S \preccurlyeq ![T]S'}$$

$$\frac{S_i \preccurlyeq S'_i \quad (\forall i \leq n)}{\&\langle l_1 : S_1, .., l_n : S_n\rangle \preccurlyeq \&\langle l_1 : S'_1, .., l_n : S'_n\rangle} \qquad \frac{S_i \preccurlyeq S'_i \quad (\forall i \leq n)}{\oplus\langle l_1 : S_1, .., l_n : S_n\rangle \preccurlyeq \oplus\langle l_1 : S'_1, .., l_n : S'_n\rangle}$$

To the system of Figure 4 we add the following rules:

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot \kappa^{+} : S \quad S' \preccurlyeq S}{\Gamma \vdash P \triangleright \Delta \cdot \kappa^{+} : S'}\ \textsc{T-PrefS} \qquad \frac{\Gamma \vdash P \triangleright \Delta \cdot \kappa^{-} : S \quad S \preccurlyeq S'}{\Gamma \vdash P \triangleright \Delta \cdot \kappa^{-} : S'}\ \textsc{T-PrefC}$$

We remark that, while rule T-PrefS would be sound with $<:$ substituted for $\preccurlyeq$ (it is just narrowing), T-PrefC is not. In fact the soundness of T-PrefC strictly depends on the fact that, differently than in the case of subtyping, & and $\oplus$ are invariant in width, while they are both covariant in depth. This is clearly connected to the following property of $\preccurlyeq$ which is the reason why Theorem 1 does not apply in the case of prefix.

Proposition 2. *For any* $S, S' \in \mathcal{ST}$, *if* $S \preccurlyeq S'$ *then* $\overline{S} \preccurlyeq \overline{S'}$.

We end the present section by establishing the basic correctness theorem for the first-order typing system. The definition of $P \equiv Q$ is the obvious extension of that of the π-calculus and can be found in [20].

Theorem 2 (Subject Reduction of the First-Order Typing System). *If $\Gamma \vdash P \triangleright \Delta$ for a (strictly) balanced Δ and $P \xrightarrow{\tau}\equiv P'$ then $\Gamma \vdash P' \triangleright \Delta'$ for some (strictly) balanced Δ'.*

4 Compliance up to Deadlock

According to [4] a client is "strongly compliant" with a service whenever it completes all direct interaction sessions with the service. On the other hand, as remarked in [7], session types do not enforce deadlock freeness in general: a client might be not strongly compliant because a deadlock occurs that prevents the session to proceed properly. As a matter of fact more is needed to guarantee deadlock freeness [12,13] or even the weaker *progress property* [5]. Since we work essentially with the original system, we can only expect a weaker concept of compliance to be warranted for typable systems, up to deadlock occurrences:

> **Weak Compliance Property:** a server cannot exhaust its actions on a channel before the corresponding client does.

To state and prove a result about weak compliance for typable systems we use some machinery to extract from a process term P the intended usage of a channel name κ^p in P, forgetting about anything else which could incidentally cause a block and prevent the full exploitation of the capabilities of P using κ^p. This is inspired to the idea of using parallel free CCS terms to describe the usage of a channel in [9,12], and to the theory of contracts [4], where CCS terms describe the protocol part of a contract.

Definition 3 (Usages). Usages *are defined by the grammar:*

$$U, V ::= 0 \mid \ell.U \mid U + V$$

where ℓ is either $\mathsf{input}, \mathsf{output}$ *or $l, \overline{l}$ for some label l; we also assume* $\overline{\mathsf{input}} = \mathsf{output}$ *and* $\overline{\mathsf{output}} = \mathsf{input}$*. Call* **Usg** *the set of usages; over* **Usg** *we define an LTS by the rules:*

$$\frac{}{\ell.U \overset{\ell}{\longmapsto} U} \qquad \frac{U \overset{\ell}{\longmapsto} U'}{U + V \overset{\ell}{\longmapsto} U'}$$

Over **Usg** *it is defined a binary relation $\preccurlyeq$:*

$$U \preccurlyeq V \quad \Leftrightarrow \quad [U \overset{\ell}{\longmapsto} U' \implies V \overset{\ell}{\longmapsto} V'].$$

We overload the notation $\preccurlyeq$ by speaking of prefix relation among usages as well as among types. Notice that we do not need to define $\preccurlyeq$ on **Usg** as a full simulation, since for our result we need only to check about the outermost action having as subject a specified channel name.

Usages and types are connected in the sense that the type of a server end κ^+ is a lower bound to its usage, while that of a client end κ^- is an upper bound, which is the contents of Theorem 3 below. To prove that we interpret session types as sets of usages, and connect the usage U to the capabilities of P restricted to the channel κ^p via the notion of *trace*.

Definition 4 (Semantics of First-Order Session Types). *Let $S \in \mathcal{ST}$; then $\llbracket S \rrbracket$ is the set of usages defined as follows:*

$$\llbracket \mathsf{end} \rrbracket = \{0\}; \quad \llbracket\, ?(T)S \rrbracket = \{\mathsf{input}.U \,|\, U \in \llbracket S \rrbracket\}; \quad \llbracket\, ![T]S \rrbracket = \{\mathsf{output}.U \,|\, U \in \llbracket S \rrbracket\}$$
$$\llbracket \&\langle l_1 : S_1, \ldots, l_n : S_n\rangle \rrbracket = \{\textstyle\sum_{i \in I} l_i.U_i \,|\, \{1, \ldots, n\} \subseteq I \wedge \forall i \in \{1, \ldots, n\}.\ U_i \in \llbracket S_i \rrbracket\};$$
$$\llbracket \oplus\langle l_1 : S_1, \ldots, l_n : S_n\rangle \rrbracket = \{\overline{l}.U \mid \exists i \in \{1, \ldots, n\}.\ l = l_i \wedge U \in \llbracket S_i \rrbracket\}.$$

By Proposition 1, we can safely abbreviate $P \xrightarrow{\tau} Q$ by $P \longrightarrow Q$. In the following $\Longrightarrow$ denotes the reflexive and transitive closure of $\longrightarrow$ and $\overset{\alpha}{\Longrightarrow}$ denotes the composition $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$. Let $\psi = \alpha_1 \cdots \alpha_n$ be in $\mathbf{Act}^*$ (ϵ will denote the empty sequence), then $P \overset{\psi}{\Longrightarrow} Q$ abbreviates $P \overset{\alpha_1}{\Longrightarrow} \cdots \overset{\alpha_n}{\Longrightarrow} Q$.

Definition 5 (Traces). *The set $Tr(P)$ of* traces *of P is defined as*

$$Tr(P) = \{\psi \in \mathbf{Act}^* \mid \exists Q.\ P \overset{\psi}{\Longrightarrow} Q\}.$$

If $\psi \in Tr(P)$ then let $\psi\restriction\kappa^p$ be the string of actions α in ψ whose subject is κ^p, and write $Tr(P, \kappa^p) = \{\psi\restriction\kappa^p \mid \psi \in Tr(P)\}$.

The mapping $\mathsf{usg} : \mathbf{Act} \rightarrow \mathbf{Usg}$ is defined by:

$$\mathsf{usg}(\kappa^p \lhd l) = \overline{l}; \qquad \mathsf{usg}(\kappa^p \rhd l) = l; \qquad \mathsf{usg}(\kappa^p(v)) = \mathsf{input}$$
$$\mathsf{usg}(\kappa^p[v]) = \mathsf{usg}(\kappa^p \nu \kappa'^q) = \mathsf{output}$$

This map extends pointwise to $\mathbf{Act}^*$ by imposing $\mathsf{usg}(\epsilon) = 0$. We shall write simply $\mathsf{usg}(\psi)$ for $\psi \in \mathbf{Act}^*$.

Lemma 1. *Let $\phi \in \mathbf{Act}^*$, then $\overline{\mathsf{usg}(\phi)} = \mathsf{usg}(\overline{\phi})$.*

Let $\mathcal{A}, \mathcal{B} \subseteq \mathbf{Usg}$, then define:

$$\mathcal{A} \sqsubseteq_* \mathcal{B} \Leftrightarrow \exists U \in \mathcal{A}.\ [U \overset{\ell}{\longmapsto} U' \implies \exists V \in \mathcal{B}.\ V \overset{\ell}{\longmapsto} V]$$
$$\mathcal{A} \sqsubseteq^* \mathcal{B} \Leftrightarrow \exists V \in \mathcal{B}\ \forall U \in \mathcal{A}.\ U \preccurlyeq V.$$

Theorem 3 (Soundness of First Order Type Interpretation). *Let $\Theta; \Gamma \vdash P \rhd \Delta$ be derivable, then:*

1. *if $\kappa^+ \in dom(\Delta)$ and $\mathcal{A} = \{\mathsf{usg}(\psi) \mid \psi \in Tr(P, \kappa^+)\}$ then $\llbracket \Delta(\kappa^+) \rrbracket \sqsubseteq_* \mathcal{A}$;*
2. *if $\kappa^- \in dom(\Delta)$ and $\mathcal{B} = \{\mathsf{usg}(\psi) \mid \psi \in Tr(P, \kappa^-)\}$ then $\mathcal{B} \sqsubseteq^* \llbracket \Delta(\kappa^-) \rrbracket$.*

Corollary 1 (Weak Compliance). *Let $\Gamma \vdash P \rhd \Delta$ be derivable for some strictly balanced Δ. If $P \overset{\alpha}{\Longrightarrow} P'$ for some P' and the subject of α is some κ^- then $\alpha'\psi' \in Tr(P, \kappa^+)$, for some ψ' and α' such that $\mathsf{usg}(\alpha') = \mathsf{usg}(\overline{\alpha})$.*

Corollary 1 does not extend straightforwardly to higher order sessions and types: this is due to the fact that the object of a throw action is associated in the typing Δ to some session type that does not correspond to any usage of the channel in that term.

5 Delegation via Higher-Order Sessions

Mobility in the π_S-calculus is formalized by the primitives throw and catch, which, respectively, send and receive channel names. According to [8], these primitives enable to implement *delegation*, that is the ability for a process to pass a session to some third party which is in charge of continuing the interaction. Such a behaviour is reflected in the typing of the subjects of throw and catch by, respectively, $![S']S$ and $?(S')S$. Adapting the rules in [8,20] to our system, where a polarized channel name κ_2^q is replaced by an unpolarized x in the body of the binding $\mathsf{catch}\ \kappa^p(x).P$, we obtain:

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot \kappa_1^p : S \quad S' \neq \mathsf{end}}{\Gamma \vdash \mathsf{throw}\ \kappa_1^p[\kappa_2^q].P \triangleright \Delta \cdot \kappa_1^p : ![S']S \cdot \kappa_2^q : S'}\ \textsc{T-Thr}'$$

$$\frac{\Gamma \vdash \{\kappa_2^q/x\}P \triangleright \Delta \cdot \kappa_1^p : S \cdot \kappa_2^q : S'}{\Gamma \vdash \mathsf{catch}\ \kappa_1^p(x).P \triangleright \Delta \cdot \kappa_1^p : ?(S')S}\ \textsc{T-Cat}'$$

where in rule T-Cat′ the implicit assumption that $\Delta \cdot \kappa_1^p : ![S']S \cdot \kappa_2^q : S'$ is well formed implies that $\kappa^q \notin \textsc{fn}(P)$.

In presence of T-PrefC, however, the above typing rules force $![S']S$ and $?(S')S$ to behave invariantly in S' w.r.t. prefix relation. In fact in rule T-Cat′ the information about the polarity of κ_2^q, while needed in the typing of $\{\kappa_2^q/x\}P$, is present in the conclusion only as input type for κ_1^p. As a consequence, any process willing to send κ_2^q via $\kappa_1^{\overline{p}}$ to $\mathsf{catch}\ \kappa_1^p(x).P$ cannot make any assumption about the actual usage of κ_2^q in $\{\kappa_2^q/x\}P$, either as a client or as a server end, and therefore the session type S' can be neither shorter (as in the case $q = +$) nor larger (when $q = -$) than the actual usage of κ_2^q.

Since invariance would be unreasonably restrictive, we need to establish a relation between the p in the conclusion and the q in the premise of rule T-Cat′. As explanatory example, let $P' =_{\text{def}} \{\kappa_2^q/x\}P$ and consider:

$$R =_{\text{def}} Q \mid \mathsf{throw}\ \kappa_1^{\overline{p}}[\kappa_2^q] \mid \mathsf{catch}\ \kappa_1^p(x)P \tag{3}$$

where $\kappa_2^{\overline{q}} \in \textsc{fn}(Q)$.

Suppose that, w.r.t. the prefix relation, $![S']S$ is contravariant in S' and covariant in S, and that $?(S')S$ is covariant both in S' and in S (this is actually the case w.r.t. subtyping I/O types in [18], and session types in [7].) Then we study under what conditions $(\nu\kappa_1)(\nu\kappa_2)R$ is safely typable, in the sense that both Weak Compliance and Error-Freeness (see [8] and Theorem 4 below) are preserved.

Let $\Gamma \vdash \mathsf{catch}\ \kappa_1^p(x)P \triangleright \Delta \cdot \kappa_1^p\ : ?(S_1)\mathsf{end}$, $\Gamma \vdash P' \triangleright \Delta \cdot \kappa_2^q : S_2$ and $\Gamma \vdash Q \triangleright \Delta' \cdot \kappa_2^{\overline{q}} : \overline{S_3}$ be derivable, and suppose that $S_1 \preccurlyeq S_2$ and $S_1 \preccurlyeq S_3$, but that S_2 and S_3 are incompatible (for example, $S_1 = ?(\mathsf{int})\mathsf{end}$, $S_2 = ?(\mathsf{int})\,![\mathsf{int}]\mathsf{end}$ and $S_3 = ?(\mathsf{int})\,![\mathsf{bool}]\mathsf{end}$).

Case $p = -$**,** $q = +$**.**

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash P' \triangleright \Delta \cdot \kappa_2^+ : S_2 \quad S_1 \preccurlyeq S_2}{\Gamma \vdash P' \triangleright \Delta \cdot \kappa_2^+ : S_1}\;\textsc{T-PrefS}}{\Gamma \vdash \mathsf{catch}\ \kappa_1^-(x)P \triangleright \Delta \cdot \kappa_1^- : ?(S_1)\mathsf{end}}\;\textsc{T-Cat}' \qquad ?(S_1)\mathsf{end} \preccurlyeq ?(S_3)\mathsf{end}}{\Gamma \vdash \mathsf{catch}\ \kappa_1^-(x)P \triangleright \Delta \cdot \kappa_1^- : ?(S_3)\mathsf{end}}\;\textsc{T-PrefC}$$

Now, since $\Gamma \vdash \mathsf{throw}\ \kappa_1^+[\kappa_2^+] \triangleright \kappa_1^+ : ![S_3]\mathsf{end} \cdot \kappa_2^+ : S_3$ by T-Thr$'$, we have that $\Gamma \vdash R \triangleright \Delta \cdot \Delta' \cdot \kappa_1^-\ ?(S_3)\mathsf{end} \cdot \kappa_1^+\ ![S_3]\mathsf{end} \cdot \kappa_2^- : \overline{S_3} \cdot \kappa_2^+ : S_3$ is derivable so that $(\nu\kappa_1)(\nu\kappa_2)R$ typechecks, but it is unsafe, as Q might require an interaction over κ_2^- which is exactly typed by $\overline{S_3}$ while P', that will receive κ_2^+, is ready to respect just the unrelated protocol S_2.
Case $p = -$**,** $q = -$: using T-PrefC with the premise $S_1 \preccurlyeq S_3$ and T-PrefS with the premise $![S_2]\mathsf{end} \preccurlyeq ![S_1]\mathsf{end}$ (implied by $S_1 \preccurlyeq S_2$ and the contravariance of $![\ ]$) we have:

$$\dfrac{\dfrac{\Gamma \vdash \mathsf{throw}\ \kappa_1^+[\kappa_2^-] \triangleright \kappa_1^+ : ![S_1]\mathsf{end} \cdot \kappa_2^- : S_1 \quad S_1 \preccurlyeq S_3}{\Gamma \vdash \mathsf{throw}\ \kappa_1^+[\kappa_2^-] \triangleright \kappa_1^+ : ![S_1]\mathsf{end} \cdot \kappa_2^- : S_3} \qquad ![S_2]\mathsf{end} \preccurlyeq ![S_1]\mathsf{end}}{\Gamma \vdash \mathsf{throw}\ \kappa_1^+[\kappa_2^-] \triangleright \kappa_1^+ : ![S_2]\mathsf{end} \cdot \kappa_2^- : S_3}$$

But since $\Gamma \vdash \mathsf{catch}\ \kappa_1^-(x)P \triangleright \Delta \cdot \kappa_1^- : ?(S_2)\mathsf{end}$ is derivable from $\Gamma \vdash P' \triangleright \Delta \cdot \kappa_2^- : S_2$ by T-Cat$'$, we have that $\Gamma \vdash R \triangleright \Delta \cdot \Delta' \cdot \kappa_1^-\ ?(S_2)\mathsf{end} \cdot \kappa_1^+\ ![S_2]\mathsf{end} \cdot \kappa_2^- : \overline{S_3} \cdot \kappa_2^+ : S_3$ is derivable but unsafe, because the server in Q will be unable to provide a service with protocol $\overline{S_2}$ as required by the client in P'.
Case $p = +$**,** $q = -$.

$$\dfrac{\dfrac{\Gamma \vdash P' \triangleright \Delta \cdot \kappa_2^- : S_2}{\Gamma \vdash \mathsf{catch}\ \kappa_1^+(x)P \triangleright \kappa_1^+ : ?(S_2)\mathsf{end}} \qquad ?(S_1)\mathsf{end} \preccurlyeq ?(S_2)\mathsf{end}}{\Gamma \vdash \mathsf{catch}\ \kappa_1^+(x)P \triangleright \kappa_1^+ : ?(S_1)\mathsf{end}}\;\textsc{T-PrefS}$$

This time we derive $\Gamma \vdash \mathsf{throw}\ \kappa_1^-[\kappa_2^-] \triangleright ![S_1]\mathsf{end} \cdot \kappa_2^- : S_3$ from $\Gamma \vdash \mathsf{throw}\ \kappa_1^-[\kappa_2^-] \triangleright ![S_1]\mathsf{end} \cdot \kappa_2^- : S_1$ by $S_1 \preccurlyeq S_3$ and T-PrefC; hence we have:

$$\Gamma \vdash R \triangleright \Delta \cdot \Delta' \cdot \kappa_1^-\ ?(S_1)\mathsf{end} \cdot \kappa_1^+\ ![S_1]\mathsf{end} \cdot \kappa_2^- : \overline{S_3} \cdot \kappa_2^+ : S_3$$

so that $(\nu\kappa_1)(\nu\kappa_2)R$ typechecks but it is unsafe since the server κ_2^+ in Q will provide a service of type $\overline{S_3}$ to the client κ_2^- in P', whose protocol is S_2.
Case $p = +$**,** $q = +$: by the previous derivation (where the polarity of κ_2 does not play any role) we have $\Gamma \vdash \mathsf{catch}\ \kappa_1^+(x)P \triangleright \kappa_1^+ : ?(S_1)\mathsf{end}$; on the other hand we have $\Gamma \vdash \mathsf{throw}\ \kappa_1^-[\kappa_2^+] \triangleright \kappa_1^- : ![S_1]\mathsf{end} \cdot \kappa_2^+ : S_3$ from $\Gamma \vdash \mathsf{throw}\ \kappa_1^-[\kappa_2^+] \triangleright \kappa_1^- : ![S_3]\mathsf{end} \cdot \kappa_2^+ : S_3$ by T-PrefC using the contravariance of $![\ ]$, which implies that $![S_3]\mathsf{end} \preccurlyeq ![S_1]\mathsf{end}$. Hence we obtain that $\Gamma \vdash R \triangleright \Delta \cdot \Delta' \cdot \kappa_1^-\ ?(S_1)\mathsf{end} \cdot \kappa_1^+\ ![S_1]\mathsf{end} \cdot \kappa_2^- : \overline{S_3} \cdot \kappa_2^+ : S_3$ with a similar mismatch as before, but having the server in P' and the client in Q.

We observe that in the cases when $p \neq q$ problems arise because of an inner incoherence of the principle of delegation for those particular client/server

$$\frac{\Gamma \vdash \{\kappa_2^p/x\}P \triangleright \Delta \cdot \kappa_1^p : S \cdot \kappa_2^p : S'}{\Gamma \vdash \mathsf{catch}\ \kappa_1^p(x).P \triangleright \Delta \cdot \kappa_1^p : ?(S')S}\ \text{T-C}\textsc{at}_p$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot \kappa_1^{\overline{p}} : S}{\Gamma \vdash \mathsf{throw}\ \kappa_1^{\overline{p}}[\kappa_2^p].P \triangleright \Delta \cdot \kappa_1^{\overline{p}} : ![S']S \cdot \kappa_2^p : S'}\ \text{T-T}\textsc{hr}_p$$

Fig. 5. The type rules for Higher-Order sessions

asymmetric interactions; namely when the throw process is a client(server) which delegates an interaction with respect to which it is working as client(server). For the cases when $p = q$, instead, the problems depend only on the contravariance of the output type ![ ]. So, a way out is to assume covariance of both input and output higher-order session types (see Definition 6) and put the equality of polarities of the subject and the object of a catch action (and consequently the duality of them in case of a throw action) into the typing rules (see Figure 5).

Definition 6 (Prefix Relation over Higher Order Session Types). *The prefix relation over $\mathcal{ST}$ is obtained by extending Definition 2 by:*

$$\frac{S_1' \preccurlyeq S_2' \quad S_1 \preccurlyeq S_2}{![S_1']S_1 \preccurlyeq ![S_2']S_2} \qquad \frac{S_1' \preccurlyeq S_2' \quad S_1 \preccurlyeq S_2}{?(S_1')S_1 \preccurlyeq ?(S_2')S_2}$$

In the rest of this section we report on results that prove the soundness of the proposed system. Session type system ensure error freeness. To define errors observe that any process term is structurally congruent to a term of the shape $(\nu\boldsymbol{\kappa})(P_1 \mid \cdots \mid P_n)$, where the P_i are prefixed processes (including branching) or selections of the shape if b then Q else R. The P_i are said to be in *head position.* If κ^p is the subject of the prefix of a process P_i we say that P_i is a *κ-process.* The parallel of dual κ-processes is a *κ-redex.*

Definition 7 (Error Freeness). *A process P is an* error *if there exists a channel κ such that either two κ-processes which do not form a κ-redex occur in P in head position, or there are more than two k-processes in head position.*

A process P is error free *if there exists no Q such that $P \xrightarrow{*} Q$ which is an error.*

The following result, proved in [8] for the original system, also holds in the asymmetric case.

Theorem 4 (Error Freeness). *If $\Gamma \vdash P \triangleright \Delta$ then P is error free.*

As said before, besides error freeness, one of the most relevant properties of asymmetric systems is the Weak Compliance Property. In presence of

Higher-order, however, this property does not hold in its full sense, as in Corollary 1. A simple counterexample can help to understand where the problem lies:

$$\mathsf{accept}\ a(y).\mathsf{request}\ b(x).\mathsf{throw}\ x[y] \ \mid\ \mathsf{request}\ a(y).y?(n) \tag{4}$$

where the process $\mathsf{request}\ a(y).y?(n)$ is a client just needing a value. The server to which such a client can connect to in order to get the needed value is $\mathsf{accept}\ a(y).\mathsf{request}\ b(x).\mathsf{throw}\ x[y]$. Such a server accepts the connection request from the client and immediately try to delegate the production of the value for the client to another server (we can look at the initial process (4) as a simplified version of the system $CalcClient \ \mid\ CalcServer_5$).

It is not difficult to check that process (4) is typable in an empty session environment, using $\{b : \uparrow[\,?(\,![\mathsf{int}]\,![\mathsf{bool}])], a : \uparrow[\,![\mathsf{int}]]\}$ as type environment. It reduces to the running process $(\nu\kappa)(\mathsf{request}\ b(x).\mathsf{throw}\ x[\kappa^+] \ \mid\ \kappa^-?(n))$ where we have a κ^--process in head position, but in which we can get an actual dual κ^+-process only in case a typable server is added to the system. Such a potentiality is represented by the presence in the system of the process $\mathsf{throw}\ x[\kappa^+]$.

The notion of "potential" κ^+-process is expressed by the following definition.

Definition 8 (Potential κ^+-process generator).
A potential κ^+-process generator *is any process of the form* $\mathsf{throw}\ k[\kappa^+].Q$

A process P is *initial* if does not contain any channel name κ neither free nor bound. A process P is *running* if there exists an initial Q such that: $Q \xrightarrow{*} P$

Theorem 5 (Higher-order Weak Compliance). *Let P be a running process which is a derivative of some typed initial process. If P contains a κ^- process in head position, then it includes either a dual κ^+-process (though not necessarily in head position) or a* potential κ^+-process generator*.*

6 Related Work and Conclusion

The main sources of this work are [8,20] and [7] on session types, and [4] for the idea of formalizing the protocols of asymmetric client/server interaction. With respect to these works we do not establish stronger results, rather we address a similar issue in a more complex setting, where processes exchange values and channels themselves, allowing for mobility and delegation.

A recent contribution which is close to our development, especially for the type interpretation, is [3]. The semantics proposed there is far more complex then ours, and different because of use of internal choice that, we think, does not model properly the label selection and its typing in the session type systems (for an attempt to establish a weak form of correspondence between contracts, that use internal choice as well, and session types see [14]).

Introducing the relation of prefix in the rules of the systems breaks the symmetry of session type systems studied so far, but surprisingly enough does not destroy the basic properties of the system, namely subject reduction and error freeness. Nonetheless a great obstacle remains, which is connected to the fact that ordinary session types do not guarantee deadlock-freeness in the sense of [10,12].

Even in presence of these limitations, we think that the system illustrated in this paper deserves interest. In fact the processes that can be represented in the π-calculus with sessions are far richer than those considered in the theory of contracts. They also are interesting w.r.t. the Service Centered Calculus (SCC) [1] and of recent proposals to detect deadlock freedom in SCC via type systems, e.g. [2].

References

1. Boreale, M., Bruni, R., Caires, L., Nicola, R.D., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V., Zavattaro, G.: SCC: a Service Centered Calculus. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 38–57. Springer, Heidelberg (2006)
2. Bruni, R., Mezzina, L.: Types and Deadlock Freedom in a Calculus of Services, Sessions and Pipelines. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 100–115. Springer, Heidelberg (2008)
3. Castagna, G., Dezani-Ciancaglini, M., Giachino, E., Padovani, L.: General Session Types (2008), `http://www.sti.uniurb.it/padovani/publications.html`
4. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. In: POPL 2008, 35th ACM Symposium on Principles of Programming Languages (January 2008)
5. Dezani-Ciancaglini, M., de' Liguoro, U., Yoshida, N.: On Progress for Structured Communications. In: Barthe, G., Fournet, C. (eds.) TGC 2007 and FODO 2008. LNCS, vol. 4912, pp. 257–275. Springer, Heidelberg (2008)
6. Gay, S., Hole, M.: Types and Subtypes for Client-Server Interactions. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 74–90. Springer, Heidelberg (1999)
7. Gay, S., Hole, M.: Subtyping for Session Types in the Pi-Calculus. Acta Informatica 42(2/3), 191–225 (2005)
8. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Disciplines for Structured Communication-based Programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
9. Igarashi, A., Kobayashi, N.: A Generic Type System for the Pi-Calculus. Theoretical Computer Science 311(1-3), 121–163 (2004)
10. Kobayashi, N.: A Type System for Lock-Free Processes. Information and Computation 177, 122–159 (2002)
11. Kobayashi, N.: Type Systems for Concurrent Programs. In: Aichernig, B.K., Maibaum, T. (eds.) Formal Methods at the Crossroads. From Panacea to Foundational Support. LNCS, vol. 2757, pp. 439–453. Springer, Heidelberg (2003)
12. Kobayashi, N.: A New Type System for Deadlock-Free Processes. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 233–247. Springer, Heidelberg (2006)
13. Kobayashi, N.: Type Systems for Concurrent Programs. Extended version of [11]. Tohoku University (2007)
14. Laneve, C., Padovani, L.: The pairing of contracts and session types. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 681–700. Springer, Heidelberg (2008)

15. Meredith, G., Bjorg, S.: Contracts and types. Commun. ACM 46(10), 41–47 (2003)
16. Parrow, J.: An introduction to the π-calculus. In: Ponse, A., Smolka, S., Bergstra, J. (eds.) Handbook of Process Algebra, ch. 8, pp. 479–544. Elsevier, Amsterdam (2001)
17. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
18. Pierce, B.C., Sangiorgi, D.: Typing and subtyping for mobile processes. In: Logic in Computer Science (1993); Full version in Mathematical Structures in Computer Science, vol. 6(5) (1996)
19. Sangiorgi, D., Walker, D.: The π-calculus. A Theory of Mobile Processes. CUP (2001)
20. Yoshida, N., Vasconcelos, V.T.: Language Primitives and Type Disciplines for Structured Communication-based Programming Revisited. In: SecReT 2006. ENTCS, vol. 171, pp. 73–93. Elsevier, Amsterdam (2007)

Equational Reasoning on Ad Hoc Networks

Fatemeh Ghassemi^1, Wan Fokkink^2, and Ali Movaghar^1

^{1} Sharif University of Technology, Tehran, Iran,
^{2} Vrije Universiteit, Amsterdam, The Netherlands
fghassemi@mehr.sharif.edu, wanf@cs.vu.nl, movaghar@sharif.edu

Abstract. We provide an equational theory for *Restricted Broadcast Process Theory* to reason about ad hoc networks. We exploit an extended algebra called *Computed Network Theory* to axiomatize restricted broadcast. It allows one to define an ad hoc network with respect to the underlying topologies. We give a sound and complete axiomatization for the recursion-free part of the term algebra *CNT*, modulo what we call rooted branching computed network bisimilarity.

1 Introduction

In Mobile Ad hoc Networks (MANETs), nodes communicate directly with each other using wireless transceivers (possibly along multihop paths) without the need for a fixed infrastructure. The primitive means of communication in MANETs is local broadcast; only nodes located in the range of a transmitter receive data. Thus nodes participate in a broadcast according to the underlying topology of nodes. On the other hand, nodes move arbitrarily, and the topology of the network changes dynamically. Local broadcast and topology changes are the main modeling challenges in MANETs.

We introduced *Restricted Broadcast Process Theory* (*RBPT*) in [6], to specify and verify ad hoc networks, taking into account mobility. *RBPT* specifies an ad hoc network by composing nodes using a restricted (local) broadcast operator, and it specifies a node by specifying a protocol deployed at a node using *RBPT* node notation. We modeled topology changes implicitly in the semantics, and thus verified a network with respect to different topology changes. An advantage of *RBPT* compared to similar algebras is that the specification of an ad hoc network does not include any specification about changes of underlying topologies. The behavior of an ad hoc network is equivalent to all its behaviors with respect to the possible topologies.

In this paper we provide an equational system to reason about *RBPT* terms. To provide equations for *RBPT* terms, we need to consider not only their observational behaviors, but also the set of topologies for which such behaviors are observed. To this aim, we first extend *RBPT* with new terms, called *Computed Network Theory* (*CNT*) because the extended terms contain a specification of a set of topologies and their observed behavior is computed with respect to those topologies. Network restrictions on the underlying topology are expressed explicitly in the syntax. The operational semantics of *CNT* is given by constrained labeled transition systems, in which the transitions are subscripted by a set of network restrictions. Our axiomatization borrows from the process algebra *ACP* [3] auxiliary (left merge and communication merge) operators to axiomatize the interleaving behavior of parallel composition.

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 113–128, 2010.

We consider an axiomatization of *CNT* modulo what we call *rooted branching computed network bisimilarity*. We prove that the axiomatization is sound, and complete for the recursion-free part of *CNT*. The application of our equational system is illustrated with a small running example.

Related works. Related calculi to ours are CBS#, CWS, CMAN, CMN, and the ω-calculus [11,10,8,9,12]. A complete comparison between ad hoc network algebras can be found in [6]. They are compared in terms of their specification and modeling concepts. In all related approaches, the only equations between networks were defined by using structural congruence. None of these papers provides a complete axiomatization for their algebra of ad hoc networks.

2 Restricted Broadcast Process Theory

Before going through the formal syntax definitions, we define some notations applied in these definitions. Let V denote a countably infinite set of variables ranged over by x, y, z, and D a finite set of data values ranged over by u. Let w range over $V \cup D$. We use $\widehat{w}$ to denote a finite sequence $w_1, \ldots, w_k$ for some $k \in I\!\!N$, $|\widehat{w}|$ its arity k, and $\{\widehat{w}/\widehat{x}\}$ for simultaneous substitutions $\{w_1/x_1\}, \ldots, \{w_k/x_k\}$. Let M denote a set of message types communicated over a network and ranged over by m, while $par : M \rightarrow I\!\!N$ defines the number of parameters encapsulated in a message m. For each message type m, there is a finite set $domain_m : I\!\!P(D^{par(m)})$ that defines the set of possible value assignments to the message parameters of m. Let Loc denote a finite set of logical addresses, ranged over by ℓ which models the hardware addresses of nodes at which protocols run. We will also use ℓ to denote a parameter of type Loc. Moreover, $A, B, C, \cdots$ denote concrete addresses. An unknown address is presented by $?$. The set of addresses extended with the unknown address is denoted as $Loc_?$, which by abuse of notation is also ranged over by ℓ.

Restricted Broadcast Process Theory (*RBPT*) [6] provides a two-level syntax to define a set of processes deployed at a node, also called protocols, and a set of ad hoc networks composed of singleton nodes:

$$P ::= 0 \mid \alpha.P \mid P + P \mid [w_1 = w_2]P, P \mid A(\widehat{w}) \,, A(\widehat{x}) \overset{def}{=} P$$

$$N ::= 0 \mid [\![P]\!]_\ell \mid N \parallel N \mid (\nu \ell)N$$

A protocol can be a deadlock, modeled by 0. $\alpha.P$ is a process that performs action α and then behaves as process P. The action α can be a send action $m(\widehat{u})!$ or a receive action $m(\widehat{x})?$. The process $P_1 + P_2$ behaves non-deterministically as process P_1 or P_2. The guarded command $[w_1 = w_2]P_1, P_2$ defines process behavior based on $w_1 = w_2$; if it evaluates to true, the protocol behaves as P_1, and otherwise as P_2. We write $A(\widehat{w})$ to denote a process invocation defined via a definition $A(\widehat{x}) \overset{def}{=} P$, with $|\widehat{x}| = |\widehat{w}|$, where $\widehat{x}$ consists of all names that appear free in P.

As a running example, $P(x) \overset{def}{=} req(x)!.P(x)$ denotes a process that broadcasts a message $req(x)$ ($par(req) = 1$ and $domain_{req} = \{0, 1\}$) recursively, and $Q \overset{def}{=} req(x)?.rep(x)!.Q$ a process that receives the message req and replies by sending

$rep(x)$ ($par(rep) = 1$ and $domain_{rep} = \{0, 1\}$) recursively. An ad hoc network can be composed of several nodes using the parallel composition operator, where each node is provided with a unique address ($\ell \neq ?$) and deploys a protocol, and nodes communicate via restricted broadcast. For instance, the network process $[\![P(0)]\!]_A \parallel [\![Q]\!]_B$ specifies an ad hoc network composed of two nodes with logical addresses A and B deploying processes $P(0)$ and Q respectively. Some address of a network can be hidden from an external observer using the restriction operator. For example, in $(\nu A)[\![P(0)]\!]_A \parallel [\![Q]\!]_B$ the activities of node A are hidden from the external observer, and only activities performed by B can be observed.

In the following section the syntax of ad hoc networks is extended with new terms, to obtain the class of what we call computed network terms. As the semantics of *RBPT* is subsumed by the one of *CNT*, we postpone an exposition on the formal semantics of *RBPT* until Section 4.

3 Computed Network Theory

As mentioned before, to give the axioms of the equational theory *RBPT*, we use an extension of *RBPT*, called *Computed Network Theory* (*CNT*). This process theory exploits network restrictions, which define a set of topologies; the behavior of process terms is computed with regard to such network restrictions.

We assume a binary relation $>$ on $Loc \times Loc_?$, which imposes connection relations between addresses. A relation $A > ?$ denotes that a node with logical address A should be in the range of there unknown address, while $A > B$ denotes a node with address A is connected to a node with address B. The relation $>$ need not be symmetric and transitive. By default each node is connected to itself: $\ell > \ell$. A *network restriction* is a set of relations $\ell > \ell'$. The network restriction $C[B/A]$ is obtained from the network restriction C by substituting B for A, and $C[B/g]$ is obtained from the network restriction C by simultaneous substitution of B for $\ell \in g$ where $g \subseteq Loc$.

A *topology* is a function $\gamma : Loc \rightarrow I\!P Loc$, where $\gamma(\ell)$ denotes the set of nodes connected to ℓ. This function models unidirectional connectivity between nodes. Each network restriction C is representative of the set of topologies that satisfy the relations in C. In particular, the empty network restriction $\{\}$ denotes all possible topologies.

CNT extends *RBPT* with new terms called computed networks, having the structure as $C\eta.\mathcal{N}$, to denote a network whose behavior, with respect to the set of topologies defined by network restriction C, is performing the action η and then behaving as $\mathcal{N}$. The parallel composition and restriction are defined over computed networks the same as *RBPT* terms. Besides *CNT* extends *RBPT* with new operators; choice ($+$), *left execution* ($\Lsh\!\!\parallel$) and *sync* ($|$):

$$\mathcal{N} ::= 0 \mid [\![P]\!]_\ell \mid C\eta.\mathcal{N} \mid \mathcal{N} + \mathcal{N} \mid \mathcal{N} \parallel \mathcal{N} \mid \mathcal{N} \,\underline{\Vert}\, \mathcal{N} \mid \mathcal{N} | \mathcal{N} \mid (\nu\ell)\mathcal{N}$$

where η can be $m(\widehat{u})!\{\ell\}$ or $m(\widehat{u})?$, and C is a network restriction. The operator $+$ defines a non-deterministic choice between *CNT* terms, and parallel composition defines computed networks communicating via restricted broadcast. The restriction operator $(\nu\ell)$ hides a node with address ℓ from an external observer as before. In left execution

$\lfloor\!\lfloor$, the left operand must perform the initial action. In the sync operator $|$, both operands perform a synchronized initial action.

Bound addresses can be α-converted, meaning that $(\nu\ell)\mathcal{N}$ equals $(\nu\ell')\mathcal{N}[\ell'/\ell]$ if $\mathcal{N}$ does not contain ℓ' as a free address. We define functions $\mathit{fl}(\mathcal{N})$ and $bl(\mathcal{N})$ to denote sets of free and bound addresses in a computed network term $\mathcal{N}$, respectively. Parameters of receive actions like $Cm(\widehat{x})?.\mathcal{N}$ are bound names in $\mathcal{N}$ while parameters of send actions like $Cm(\widehat{x})!\{\ell\}.\mathcal{N}$ are free names in $\mathcal{N}$. A computed network term is called closed if its set of free names is empty.

4 Operational Semantics of *CNT*

The operational semantics of *CNT* is given at two levels (similar to the syntax), in terms of the operational semantics of protocols and of computed network processes.

Given a protocol, the operational rules of Table 1 induce a labeled transition system, in which the transitions are of the form $P \xrightarrow{\alpha} P'$ with $\alpha \in \{m(\widehat{u})?, m(\widehat{u})!\}$. They are standard operational rules for basic process algebras. (For explanations about the protocol operational rules, the reader is referred to [6].)

Table 1. Semantics of protocols

$$\frac{}{m(\widehat{x})?.P \xrightarrow{m(\widehat{u})?} P\{\widehat{u}/\widehat{x}\}} : \mathit{Pre}_1 \qquad \frac{}{m(\widehat{u})!.P \xrightarrow{m(\widehat{u})!} P} : \mathit{Pre}_2$$

$$\frac{P\{\widehat{u}/\widehat{x}\} \xrightarrow{\alpha} P'}{A(\widehat{u}) \xrightarrow{\alpha} P'} : \mathit{Inv},\ A(x) \overset{\mathit{def}}{=} P \qquad \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 + P_2 \xrightarrow{\alpha} P_1'} : \mathit{Choice}$$

$$\frac{P_1 \xrightarrow{\alpha} P_1'}{[u = u]P_1, P_2 \xrightarrow{\alpha} P_1'} : \mathit{Then} \qquad \frac{P_2 \xrightarrow{\alpha} P_2'}{[u_1 = u_2]P_1, P_2 \xrightarrow{\alpha} P_2'} : \mathit{Else},\ u_1 \neq u_2$$

Generally the behavior of a computed network is defined in terms of a set of topologies; a transition, in which a set of nodes participate in a communication, is possible for all topologies in which the receiving nodes are connected to the sending node. Therefore in the operational semantics it is defined for each state which transitions are possible for which sets of topologies (out of all possible topologies). Network restrictions are used to define the set of underlying topologies for each transition.

Given a computed network, the operational rules in Table 2 induce a constrained labeled transition system of transitions $\mathcal{N} \xrightarrow{\eta}_C \mathcal{N}'$, where C is a network restriction defining a set of topologies under which this transition is possible, and η can be a send or receive. The operational rules of computed networks are shown in Table 2. The symmetric counterparts of rules Choice', Bro, Sync_2 and Par have been omitted. In this table $\mathit{hide}(C, \ell)$ denotes $\{\ell_1 > \ell_2 | \ell_1 > \ell_2 \in C[?/\ell] \wedge \ell_1 \neq ?\}$. Moreover, $\eta[\ell'/\ell]$ denotes η with all occurrences of ℓ replaced by ℓ'.

Inter_1 denotes that a single node can perform the send actions of a protocol at this node under any valid topology, and its network address is appended to this action. Inter_2 denotes a single node performing a receive action, under the restriction that

Table 2. Semantics of *CNT* terms

$$\frac{P \xrightarrow{m(\widehat{u})!} P'}{[\![P]\!]_\ell \xrightarrow{m(\widehat{u})!\{\ell\}}_{\{\}} [\![P']\!]_\ell} : Inter_1 \qquad \frac{P \xrightarrow{m(\widehat{u})?} P'}{[\![P]\!]_\ell \xrightarrow{m(\widehat{u})?}_{\{\ell>?\}} [\![P']\!]_\ell} : Inter_2$$

$$\frac{}{C\eta.\mathcal{N} \xrightarrow{\eta}_C \mathcal{N}} : Pre' \qquad \frac{\mathcal{N}_1 \xrightarrow{\eta}_C \mathcal{N}_1'}{\mathcal{N}_1 + \mathcal{N}_2 \xrightarrow{\eta}_C \mathcal{N}_1'} : Choice'$$

$$\frac{\mathcal{N} \xrightarrow{\eta}_C \mathcal{N}'}{\mathcal{N} \xrightarrow{\eta}_{C'} \mathcal{N}'} : \; Exe, \quad C \subseteq C' \qquad \frac{\mathcal{N}_1 \xrightarrow{m(\widehat{u})?}_{C_1} \mathcal{N}_1' \;\; \mathcal{N}_2 \xrightarrow{m(\widehat{u})?}_{C_2} \mathcal{N}_2'}{\mathcal{N}_1 \parallel \mathcal{N}_2 \xrightarrow{m(\widehat{u})?}_{C_1 \cup C_2} \mathcal{N}_1' \parallel \mathcal{N}_2'} : Recv$$

$$\frac{\mathcal{N}_1 \xrightarrow{m(\widehat{u})!\{\ell\}}_{C_1} \mathcal{N}_1' \;\; \mathcal{N}_2 \xrightarrow{m(\widehat{u})?}_{C_2} \mathcal{N}_2'}{\mathcal{N}_1 \parallel \mathcal{N}_2 \xrightarrow{m(\widehat{u})!\{\ell\}}_{C_1 \cup C_2[\ell/?]} \mathcal{N}_1' \parallel \mathcal{N}_2'} : Bro \qquad \frac{\mathcal{N}_1 \xrightarrow{\eta}_C \mathcal{N}_1'}{\mathcal{N}_1 \parallel \mathcal{N}_2 \xrightarrow{\eta}_C \mathcal{N}_1' \parallel \mathcal{N}_2} : Par$$

$$\frac{\mathcal{N}_1 \xrightarrow{m(\widehat{u})?}_{C_1} \mathcal{N}_1' \;\;\; \mathcal{N}_2 \xrightarrow{m(\widehat{u})?}_{C_2} \mathcal{N}_2'}{\mathcal{N}_1 \mid \mathcal{N}_2 \xrightarrow{m(\widehat{u})?}_{C_1 \cup C_2} \mathcal{N}_1' \parallel \mathcal{N}_2'} : Sync_1 \qquad \frac{\mathcal{N}_1 \xrightarrow{\eta}_C \mathcal{N}_1'}{\mathcal{N}_1 \mathbin{\lfloor\!\lfloor} \mathcal{N}_2 \xrightarrow{\eta}_C \mathcal{N}_1' \parallel \mathcal{N}_2} : \; LExe$$

$$\frac{\mathcal{N}_1 \xrightarrow{m(\widehat{u})!\{\ell\}}_{C_1} \mathcal{N}_1' \;\; \mathcal{N}_2 \xrightarrow{m(\widehat{u})?}_{C_2} \mathcal{N}_2'}{\mathcal{N}_1 \mid \mathcal{N}_2 \xrightarrow{m(\widehat{u})!\{\ell\}}_{C_1 \cup C_2[\ell/?]} \mathcal{N}_1' \parallel \mathcal{N}_2'} : Sync_2 \qquad \frac{\mathcal{N} \xrightarrow{\eta}_C \mathcal{N}'}{(\nu\ell)\mathcal{N} \xrightarrow{\eta[?/\ell]}_{hide(C,\ell)} (\nu\ell)\mathcal{N}'} : Rest$$

the node must be connected to some sender (denoted by $?$) is added to the network restriction. Pre' indicates execution of a prefix action. $Choice'$ defines that a computed network can behave non-deterministically. Exe indicates that if a transition is possible for C, then it is also possible for any more restrictive C'. $Recv$ allows to group together nodes that are ready to receive the same message. Bro indicates the actual synchronization in local broadcast among a transmitter and receivers. This transition is valid for all topologies in which the transmitter is connected (not essentially bidirectly) to the receivers, which is captured by $C_1 \cup C_2[\ell/?]$. The communication results in a transition labeled with $m(\widehat{u})!\{\ell\}$, so the message $m(\widehat{u})!$ remains visible to be received by other computed networks.

We consider a possible transition of the running example introduced in Section 2. This transition, given below, results from applications of $Inter_1$, $Inter_2$ and Bro:

$$[\![P(0)]\!]_A \parallel [\![Q]\!]_B \xrightarrow{req(0)!\{A\}}_{\{B>A\}} [\![P(0)]\!]_A \parallel [\![rep(0)!.Q]\!]_B$$

In this transition, node A broadcasts a message $req(0)$ and node B receives it, so that the parameter x is substituted by 0. This transition is possible for topologies in which B is connected to A, i.e. the accompanying network restriction is $\{B > A\}$.

As the sync operator defines synchronization between two computed networks, its behavior is defined by $Sync_1$ and $Sync_2$ indicating synchronization on a receive action (sent by the context) or a communication. $LExe$ defines that in a term composed by the left execution, the left computed network performs the initial action, and then the resulting term proceeds as in parallel composition. Par defines locality for a computed network; an event in a computed network may result from this same event in a subnetwork.

Another possible transition of $[\![P(0)]\!]_A \parallel [\![Q]\!]_B$, resulting from an application of $Inter_1$ and Par, is:

$$[\![P(0)]\!]_A \parallel [\![Q]\!]_B \xrightarrow{req(0)!\{A\}}_{\{\}} [\![P(0)]\!]_A \parallel [\![Q]\!]_B$$

In this transition, node A sends but B does not participate in communication. This transition is possible for all possible topologies (so B may be connected to A, but it has lost the message), denoted by $\{\}$.

$Rest$ makes sure that restrictions over invisible addresses are removed and the address of a sender with hidden address is concealed from the external observer by converting its address to $?$. By using network restrictions, we can easily define the set of topologies over visible nodes under which such a transition is possible (by removing restrictions imposed on hidden nodes).

In the running example, if we hide node A, then the possible transitions when A broadcasts (resulting from $Inter_{1,2}$, $Rest$, Bro or Par) are:

$$(\nu A)[\![P(0)]\!]_A \| [\![Q]\!]_B \xrightarrow{req(0)!\{?\}}_{\{B>?\}} (\nu A)[\![P(0)]\!]_A \parallel [\![rep(0)!.Q]\!]_B$$

$$(\nu A)[\![P(0)]\!]_A \parallel [\![Q]\!]_B \xrightarrow{req(0)!\{?\}}_{\{\}} (\nu A)[\![P(0)]\!]_A \parallel [\![Q]\!]_B.$$

Here the observer cannot see who has performed the send action.

5 Computed Network Bisimulation

We define the notion of computed network bisimilarity between nodes in a constrained labeled transition system, based on the notion of branching bisimilarity [13]. Our observer is distributed over locations of nodes with visible addresses equipped with a sensor to sense signals (and decrypt the signals in wireless communications). If the strength of a signal at a node is of a predefined threshold, it concludes that the node has performed a send action. If it cannot conclude the sender of the message, it will consider it as a send action with an unknown sender. To define our observational equivalence relation, we introduce the following notations:

- $\Rightarrow$ denotes the reflexive and transitive closure of receive actions which preserve topologies:
  - $\mathcal{N} \Rightarrow \mathcal{N}$;
  - if $\mathcal{N} \xrightarrow{m(\widehat{u})?}_{\{\}} \mathcal{N}'$ and $\mathcal{N}' \Rightarrow \mathcal{N}''$, then $\mathcal{N} \Rightarrow \mathcal{N}''$.
- $\xrightarrow{\overline{\eta}}_C$ denotes that either $\xrightarrow{\eta}_C$, or η is of the form $m(\widehat{u})!\{?\}$ and $\xrightarrow{\eta[\ell/?]}_{C[\ell/?]}$.

Definition 1. *A binary relation $\mathcal{R}$ on computed network terms is a branching computed network simulation, if $\mathcal{N}_1 \mathcal{R} \mathcal{N}_2$ implies whenever $\mathcal{N}_1 \xrightarrow{\eta}_C \mathcal{N}_1'$:*

- *η is of the form $m(\widehat{u})?$, and $\mathcal{N}_1' \mathcal{R} \mathcal{N}_2$;*
- *or there are $\mathcal{N}_2'$ and $\mathcal{N}_2''$ such that $\mathcal{N}_2 \Rightarrow \mathcal{N}_2'' \xrightarrow{\overline{\eta}}_C \mathcal{N}_2'$, where $\mathcal{N}_1 \mathcal{R} \mathcal{N}_2''$ and $\mathcal{N}_1' \mathcal{R} \mathcal{N}_2'$.*

$\mathcal{R}$ is a branching computed network bisimulation if $\mathcal{R}$ and $\mathcal{R}^{-1}$ are branching computed network simulations. Computed networks $\mathcal{N}_1$ and $\mathcal{N}_2$ are branching computed network bisimilar, written $\mathcal{N}_1 \simeq_b \mathcal{N}_2$, if $\mathcal{N}_1 \mathcal{R} \mathcal{N}_2$ for some branching computed network bisimulation relation $\mathcal{R}$.

Computed network bisimilarity is not a congruence with respect to the choice operator. To obtain a congruence, we need to add a root condition.

Definition 2. *Two computed networks $\mathcal{N}_1$ and $\mathcal{N}_2$ are* rooted branching computed network bisimilar, *written $\mathcal{N}_1 \simeq_{rb} \mathcal{N}_2$,*

- *if $\mathcal{N}_1 \xrightarrow{\eta}_C \mathcal{N}_1'$, then there is an $\mathcal{N}_2'$ such that $\mathcal{N}_2 \xrightarrow{\overline{\eta}}_C \mathcal{N}_2'$, and $\mathcal{N}_1' \simeq_b \mathcal{N}_2'$;*
- *if $\mathcal{N}_2 \xrightarrow{\eta}_C \mathcal{N}_2'$, then there is an $\mathcal{N}_1'$ such that $\mathcal{N}_1 \xrightarrow{\overline{\eta}}_C \mathcal{N}_1'$, and $\mathcal{N}_1' \simeq_b \mathcal{N}_2'$.*

We proved that branching computed network bisimilarity and rooted branching computed network bisimilarity are equivalence relations over computed networks. Moreover, the latter constitutes a congruence with respect to *CNT*. See Appendix A and B.

6 *CNT* Axiomatization

Our axiomatization for *CNT* terms is given in Table 3. P_{1-8} axiomatize protocols deployed at a node. In P_4, summation $\sum$ is used to denote a choice over a finite set, in this case $domain_m$; summation over an empty set denotes 0. *Dead* explains that hiding an address in a deadlock computed network has no affect. *Con* expresses that when a same behavior happens under two different sets of topologies, and if one set is included in another set, then from the point view of an external observer, the behavior occurs for the superset of topologies. Obs expresses when a send from a hidden action has no effect and can be equated to any send from a visible action. Cho_{1-4} define idempotency, commutativity, associativity and unit element for the choice operator. The parallel composition of two computed network is defined in an interleaving semantics the same as in the process algebra *ACP* [3] by the axiom *Br*; in a network composed of two computed networks $\mathcal{N}_1$ and $\mathcal{N}_2$, each network may perform a local action, or they may have communication via local broadcast. LEx_{1-3} define axioms for left execution; in left execution, the left operand performs an action (LEx_1), choice operator can be distributed over left execution (LEx_2), and when the left operand cannot do any action, then left execution results into a deadlock (LEx_3). S_1 and S_2 define commutativity and distributivity of choice over the sync operator, respectively. S_3 defines that when an argument in a sync composition is a deadlock, then the result of sync composition is a deadlock. $Sync_{1-5}$ define synchronization between two computed network terms. Generally speaking, two terms can be synchronized if they send/receive the same message with the same parameter values. T_1 and T_2 express when a receive action can be removed. Res_1 defines scope extrusion of the restriction operator. $Res_{2,4}$ define that the order and number of repeats of the restriction operator have no effect on the behavior of computed network terms. Res_3 defines distribution of restriction over the choice operator. Res_{5-7} express the effect of the restriction operator: network restrictions over hidden addresses are removed. In Res_5, restriction has no effect on send actions from visible addresses, except for removing restrictions over hidden addresses. In Res_6, the address of a hidden sender is converted to $?$.

Table 3. Axiomatization of *CNT* terms

$\llbracket 0 \rrbracket_\ell = 0$	P_1	$\llbracket m(\widehat{u})!.P \rrbracket_\ell = \{\}m(\widehat{u})!\{\ell\}.\llbracket P \rrbracket_\ell$	P_2
$\llbracket m(\widehat{u})?.P \rrbracket_\ell = \{\ell >?\}m(u)?.\llbracket P \rrbracket_\ell$	P_3	$\llbracket m(\widehat{y})?.P \rrbracket_\ell = \sum_{\widehat{u} \in domain_m} \llbracket m(\widehat{u})?.P[\widehat{u}/\widehat{y}] \rrbracket_\ell$	P_4
$\llbracket P_1 + P_2 \rrbracket_\ell = \llbracket P_1 \rrbracket_\ell + \llbracket P_2 \rrbracket_\ell$	P_5	$\llbracket A(\widehat{u}) \rrbracket_\ell = \llbracket P[\widehat{u}/\widehat{x}] \rrbracket_\ell, A(\widehat{x}) \overset{def}{=} P$	P_6
$\llbracket [u = u]P_1, P_2 \rrbracket_\ell = \llbracket P_1 \rrbracket_\ell$	P_7	$\llbracket [u_1 = u_2]P_1, P_2 \rrbracket_\ell = \llbracket P_2 \rrbracket_\ell \qquad (u_1 \neq u_2)$	P_8
$0 = (\nu\ell)0$	*Dead*		
$C_1\eta.\mathcal{N} + C_2\eta.\mathcal{N} = C_1\eta.\mathcal{N} \qquad (C_1 \subseteq C_2)$	*Con*		
$Cm(\widehat{u})!\{?\}.\mathcal{N} + C[\ell/?]m(\widehat{u})!\{\ell\}.\mathcal{N} = C[\ell/?]m(\widehat{u})!\{\ell\}.\mathcal{N}$	*Obs*		
$\mathcal{N} + \mathcal{N} = \mathcal{N}$	Cho_1	$\mathcal{N}_1 + (\mathcal{N}_2 + \mathcal{N}_3) = (\mathcal{N}_1 + \mathcal{N}_2) + \mathcal{N}_3$	Cho_3
$\mathcal{N}_1 + \mathcal{N}_2 = \mathcal{N}_2 + \mathcal{N}_1$	Cho_2	$\mathcal{N} + 0 = \mathcal{N}$	Cho_4
$\mathcal{N}_1 \parallel \mathcal{N}_2 = \mathcal{N}_1 \mathbin{\parallel\!\!\lfloor} \mathcal{N}_2 + \mathcal{N}_2 \mathbin{\parallel\!\!\lfloor} \mathcal{N}_1 + \mathcal{N}_1 \mid \mathcal{N}_2$	Br		
$C\eta.\mathcal{N}_1 \mathbin{\parallel\!\!\lfloor} \mathcal{N}_2 = C\eta.(\mathcal{N}_1 \parallel \mathcal{N}_2)$	LEx_1		
$(\mathcal{N}_1 + \mathcal{N}_2) \mathbin{\parallel\!\!\lfloor} \mathcal{N} = \mathcal{N}_1 \mathbin{\parallel\!\!\lfloor} \mathcal{N} + \mathcal{N}_2 \mathbin{\parallel\!\!\lfloor} \mathcal{N}$	LEx_2		
$0 \mathbin{\parallel\!\!\lfloor} \mathcal{N} = 0$	LEx_3		
$\mathcal{N}_1 \mid \mathcal{N}_2 = \mathcal{N}_2 \mid \mathcal{N}_1$	S_1		
$(\mathcal{N}_1 + \mathcal{N}_2) \mid \mathcal{N} = \mathcal{N}_1 \mid \mathcal{N} + \mathcal{N}_2 \mid \mathcal{N}$	S_2		
$0 \mid \mathcal{N} = 0$	S_3		
$C_1m(\widehat{u})!\{\ell\}.\mathcal{N}_1 \mid C_2m(\widehat{u})?.\mathcal{N}_2 = C_1 \cup C_2[\ell/?]m(\widehat{u})!\{\ell\}.(\mathcal{N}_1 \parallel \mathcal{N}_2)$	$Sync_1$		
$C_1m(\widehat{u}_1)!\{\ell\}.\mathcal{N}_1 \mid C_2n(\widehat{u}_2)?.\mathcal{N}_2 = 0 \qquad (m \neq n \vee \widehat{u}_1 \neq \widehat{u}_2)$	$Sync_2$		
$C_1m(\widehat{u})?.\mathcal{N}_1 \mid C_2m(\widehat{u})?.\mathcal{N}_2 = C_1 \cup C_2m(\widehat{u})?.(\mathcal{N}_1 \parallel \mathcal{N}_2)$	$Sync_3$		
$C_1m(\widehat{u}_1)?.\mathcal{N}_1 \mid C_2n(\widehat{u}_2)?.\mathcal{N}_2 = 0 \qquad (m \neq n \vee \widehat{u}_1 \neq \widehat{u}_2)$	$Sync_4$		
$C_1m(\widehat{u}_1)!.\mathcal{N}_1\{\ell_1\} \mid C_2n(\widehat{u}_2)!\{\ell_2\}.\mathcal{N}_2 = 0$	$Sync_5$		
$C\eta.(C'm(\widehat{u})?.\mathcal{N} + \mathcal{N}) = C\eta.\mathcal{N}$	T_1		
$C\eta.(\{\}m(\widehat{u})?.(\mathcal{N}_1 + \mathcal{N}_2) + \mathcal{N}_2) = C\eta.(\mathcal{N}_1 + \mathcal{N}_2)$	T_2		
$(\nu\ell)\mathcal{N}_1 \parallel \mathcal{N}_2 = (\nu\ell)(\mathcal{N}_1 \parallel \mathcal{N}_2) \quad (\ell \not\in fl(\mathcal{N}_2))$	Res_1	$(\nu\ell_1)(\nu\ell_2)\mathcal{N} = (\nu\ell_2)(\nu\ell_1)\mathcal{N}$	Res_2
$(\nu\ell)(\mathcal{N}_1 + \mathcal{N}_2) = (\nu\ell)\mathcal{N}_1 + (\nu\ell)\mathcal{N}_2$	Res_3	$(\nu\ell)\mathcal{N} = \mathcal{N} \qquad (\ell \not\in fl(\mathcal{N}))$	Res_4
$(\nu\ell)Cm(\widehat{u})!\{\ell'\}.\mathcal{N} = hide(C,\ell)m(\widehat{u})!\{\ell'\}.(\nu\ell)\mathcal{N} \qquad (\ell \neq \ell')$	Res_5		
$(\nu\ell)Cm(\widehat{u})!\{\ell\}.\mathcal{N} = hide(C,\ell)m(\widehat{u})!\{?\}.(\nu\ell)\mathcal{N}$	Res_6		
$(\nu\ell)Cm(\widehat{u})?.\mathcal{N} = hide(C,\ell)m(\widehat{u})?.(\nu\ell)\mathcal{N}$	Res_7		

Theorem 1. *CNT is a sound axiomatization of the term algebra $\mathbb{P}(CNT)/\simeq_{rb}$, i.e. for all closed computed network terms $\mathcal{N}_1$ and $\mathcal{N}_2$, if $\mathcal{N}_1 = \mathcal{N}_2$ then $\mathcal{N}_1 \simeq_{rb} \mathcal{N}_2$.*

Theorem 2. *CNT is a complete axiomatization for the recursion-free part of the term algebra $\mathbb{P}(CNT)/\simeq_{rb}$, i.e. for all closed, recursion-free computed network terms $\mathcal{N}_1$ and $\mathcal{N}_2$, $\mathcal{N}_1 \simeq_{rb} \mathcal{N}_2$ implies $\mathcal{N}_1 = \mathcal{N}_2$.*

We prove this theorem in [7] using a restricted graph model which is isomorphic to the term algebra $\mathbb{P}(CNT)/\simeq_{rb}$, following the approach of [4,13,1]. The basic idea in the completeness proof is to establish a graph rewriting system on restricted graphs, which is confluent and strongly normalizing (up to restricted graph isomorphism), and for which every rewrite step preserves rooted branching graph bisimilarity. Then we prove that a rewrite step can be mapped to a proof step in *CNT*. By finding an identity relation between functions relating graphs and *CNT* terms, completeness can be concluded. The identity relation can be easily proved for basic terms; a basic term only consists of prefix and choice operators. Axioms in Table 3 allow us to bring all recursion-free closed *CNT* terms in basic terms.

We apply the axioms in Table 3 to the running example.

$$
\begin{aligned}
&\llbracket P(0)\rrbracket_A = \{\}\mathit{req}(0)!\{A\}.\llbracket P(0)\rrbracket_A\\
&\llbracket Q\rrbracket_B = \textstyle\sum_{i=0,1}\{B>?\}\mathit{req}(i)?.\llbracket \mathit{rep}(i)!.Q\rrbracket_B\\
&\llbracket P(0)\rrbracket_A \parallel \llbracket Q\rrbracket_B = \llbracket P(0)\rrbracket_A \mathbin{\lfloor\!\lfloor} \llbracket Q\rrbracket_B + \llbracket Q\rrbracket_B \mathbin{\lfloor\!\lfloor} \llbracket P(0)\rrbracket_A + \llbracket P(0)\rrbracket_A \mid \llbracket Q\rrbracket_B\\
&\ = \{\}\mathit{req}(0)!\{A\}.\llbracket P(0)\rrbracket_A \mathbin{\lfloor\!\lfloor} \llbracket Q\rrbracket_B + \textstyle\sum_{i=0,1}\{B>?\}\mathit{req}(i)?.\llbracket \mathit{rep}(i)!.Q\rrbracket_B \mathbin{\lfloor\!\lfloor} \llbracket P(0)\rrbracket_A\\
&\quad + \{\}\mathit{req}(0)!\{A\}.\llbracket P(0)\rrbracket_A \mid \textstyle\sum_{i=0,1}\{B>?\}\mathit{req}(i)?.\llbracket \mathit{rep}(i)!.Q\rrbracket_B\\
&\ = \{\}\mathit{req}(0)!\{A\}.\llbracket P(0)\rrbracket_A \parallel \llbracket Q\rrbracket_B + \textstyle\sum_{i=0,1}\{B>?\}\mathit{req}(i)?.\llbracket P(0)\rrbracket_A \parallel \llbracket \mathit{rep}(i)!.Q\rrbracket_B\\
&\quad + \{B>A\}\mathit{req}(0)!\{A\}.\llbracket P(0)\rrbracket_A \parallel \llbracket \mathit{rep}(0)!.Q\rrbracket_B
\end{aligned}
$$

indicating that the behavior can be: A can broadcast a message but B does not participate, or B can receive a message sent by its context, or A can broadcast a message and B receives it for a set of topologies in which B is connected to A.

Now let C be a hidden node with a behavior like A:

$$
\begin{aligned}
&\llbracket P(0)\rrbracket_A \parallel \llbracket Q\rrbracket_B \parallel (\nu C)\llbracket P(0)\rrbracket_C = (\nu C)(\llbracket P(0)\rrbracket_A \parallel \llbracket Q\rrbracket_B \parallel \llbracket P(0)\rrbracket_C)\\
&\ = (\nu C)(\{\}\mathit{req}(0)!\{A\}.\llbracket P(0)\rrbracket_A \parallel \llbracket Q\rrbracket_B \parallel \llbracket P(0)\rrbracket_C\\
&\qquad + \{\}\mathit{req}(0)!\{C\}.\llbracket P(0)\rrbracket_A \parallel \llbracket Q\rrbracket_B \parallel \llbracket P(0)\rrbracket_C\\
&\qquad + \textstyle\sum_{i=0,1}\{B>?\}\mathit{req}(i)?.\llbracket P(0)\rrbracket_A \parallel \llbracket \mathit{rep}(i)!.Q\rrbracket_B \parallel \llbracket P(0)\rrbracket_C\\
&\qquad + \{B>A\}\mathit{req}(0)!\{A\}.\llbracket P(0)\rrbracket_A \parallel \llbracket \mathit{rep}(0)!.Q\rrbracket_B \parallel \llbracket P(0)\rrbracket_C\\
&\qquad + \{B>C\}\mathit{req}(0)!\{C\}.\llbracket P(0)\rrbracket_A \parallel \llbracket \mathit{rep}(0)!.Q\rrbracket_B \parallel \llbracket P(0)\rrbracket_C)\\
&\ = \{\}\mathit{req}(0)!\{A\}.(\nu C)(\llbracket P(0)\rrbracket_A \parallel \llbracket Q\rrbracket_B \parallel \llbracket P(0)\rrbracket_C)\\
&\quad + \{\}\mathit{req}(0)!\{?\}.(\nu C)(\llbracket P(0)\rrbracket_A \parallel \llbracket Q\rrbracket_B \parallel \llbracket P(0)\rrbracket_C)\\
&\quad + \textstyle\sum_{i=0,1}\{B>?\}\mathit{req}(i)?.(\nu C)(\llbracket P(0)\rrbracket_A \parallel \llbracket \mathit{rep}(i)!.Q\rrbracket_B \parallel \llbracket P(0)\rrbracket_C)\\
&\quad + \{B>A\}\mathit{req}(0)!\{A\}.(\nu C)(\llbracket P(0)\rrbracket_A \parallel \llbracket \mathit{rep}(0)!.Q\rrbracket_B \parallel \llbracket P(0)\rrbracket_C)\\
&\quad + \{B>?\}\mathit{req}(0)!\{?\}.(\nu C)(\llbracket P(0)\rrbracket_A \parallel \llbracket \mathit{rep}(0)!.Q\rrbracket_B \parallel \llbracket P(0)\rrbracket_C)\\
&\ = \{\}\mathit{req}(0)!\{A\}.(\nu C)(\llbracket P(0)\rrbracket_A \parallel \llbracket Q\rrbracket_B \parallel \llbracket P(0)\rrbracket_C)\\
&\quad + \textstyle\sum_{i=0,1}\{B>?\}\mathit{req}(i)?.(\nu C)(\llbracket P(0)\rrbracket_A \parallel \llbracket \mathit{rep}(i)!.Q\rrbracket_B \parallel \llbracket P(0)\rrbracket_C)\\
&\quad + \{B>A\}\mathit{req}(0)!\{A\}.(\nu C)(\llbracket P(0)\rrbracket_A \parallel \llbracket \mathit{rep}(0)!.Q\rrbracket_B \parallel \llbracket P(0)\rrbracket_C)
\end{aligned}
$$

We can derive $\llbracket P(0)\rrbracket_A \parallel \llbracket Q\rrbracket_B = \llbracket P(0)\rrbracket_A \parallel \llbracket Q\rrbracket_B \parallel (\nu C)\llbracket P(0)\rrbracket_C$, as the following equality holds:

$$\llbracket P(0)\rrbracket_A \parallel \llbracket \mathit{rep}(i)!.Q\rrbracket_B \parallel (\nu C)\llbracket P(0)\rrbracket_C = \llbracket P(0)\rrbracket_A \parallel \llbracket \mathit{rep}(i)!.Q\rrbracket_B.$$

Now consider a protocol called $R(x)$, which can send the request x or receive a request. When it receives a request y, it either replies by sending the request y, or ignores it and waits until it receives that request again. The definition of this protocol is

$$\begin{aligned}
R(x) &\stackrel{def}{=} \mathit{req}(x)!.R(x) + \mathit{req}(x)!.S(x) + \mathit{req}(y)?.Z(x,y)\\
S(x) &\stackrel{def}{=} \mathit{req}(x)!.S(x) + \mathit{rep}(x)!.R(x)\\
Z(x,y) &\stackrel{def}{=} \mathit{req}(y)?.Z(x,y) + \mathit{rep}(y)!.R(x) + \mathit{req}(x)!.Z(x,y)
\end{aligned}$$

The behavior of a network consisting of a hidden node B, with protocol $R(0)$ deployed, is:

$$[\![Z(0,i)]\!]_B = \{B>?\}\mathit{req}(i)?.[\![Z(0,i)]\!]_B + \{\}\mathit{rep}(i)!\{B\}.[\![R(0)]\!]_B + \{\}\mathit{req}(0)!\{B\}.[\![Z(0,i)]\!]_B$$

$$\begin{aligned}
(\nu B)[\![R(0)]\!]_B &= (\nu B)(\{\}\mathit{req}(0)!\{B\}.[\![R(0)]\!]_B + \{\}\mathit{req}(0)!\{B\}.[\![S(0)]\!]_B\\
&\quad + \textstyle\sum_{i=0,1}\{B>?\}\mathit{req}(i)?.[\![Z(0,i)]\!]_B)\\
&= \{\}\mathit{req}(0)!\{?\}.(\nu B)[\![R(0)]\!]_B + \{\}\mathit{req}(0)!\{?\}.[\![S(0)]\!]_B\\
&\quad + \textstyle\sum_{i=0,1}\{\}\mathit{req}(i)?.(\nu B)[\![Z(0,i)]\!]_B
\end{aligned}$$

We can derive $(\nu A,B)[\![P(0)]\!]_A \parallel [\![Q]\!]_B = (\nu B)[\![R(0)]\!]_B$, as the following equalities hold:

$$\begin{aligned}
&(\nu A,B)([\![P(0)]\!]_A \parallel [\![\mathit{rep}(0)!.Q]\!]_B) = (\nu A,B)([\![S(0)]\!]_B)\\
&\{\}\mathit{req}(i)?.(\nu A,B)([\![P(0)]\!]_A \parallel [\![\mathit{rep}(i)!.Q]\!]_B) = \{\}\mathit{req}(i)?.(\nu B)[\![Z(0,i)]\!]_B.
\end{aligned}$$

For instance, $\{\}\mathit{req}(i)?.(\nu A,B)([\![P(0)]\!]_A \parallel [\![\mathit{rep}(i)!.Q]\!]_B) = \{\}\mathit{req}(i)?.(\nu B)[\![Z(0,i)]\!]_B$ holds as:

$$\begin{aligned}
&\{\}\mathit{req}(i)?.(\nu B)[\![Z(0,i)]\!]_B =\\
&\qquad \{\}\mathit{req}(i)?.(\{\}\mathit{req}(i)?.(\nu B)[\![Z(0,i)]\!]_B + \{\}\mathit{rep}(i)!\{?\}.(\nu B)[\![R(0)]\!]_B\\
&\qquad +\{\}\mathit{req}(0)!\{?\}.(\nu B)[\![Z(0,i)]\!]_B) =\\
&\qquad \{\}\mathit{req}(i)?.((\nu B)[\![Z(0,i)]\!]_B + \{\}\mathit{rep}(i)!\{?\}.(\nu B)[\![R(0)]\!]_B\\
&\qquad +\{\}\mathit{req}(0)!\{?\}.(\nu B)[\![Z(0,i)]\!]_B)
\end{aligned}$$

$$\begin{aligned}
&\{\}\mathit{req}(i)?.(\nu A,B)([\![P(0)]\!]_A \parallel [\![\mathit{rep}(i)!.Q]\!]_B)\\
&\qquad = \{\}\mathit{req}(i)?.(\{\}\mathit{rep}(i)!\{?\}.(\nu A,B)([\![P(0)]\!]_A \parallel [\![Q]\!]_B)\\
&\qquad +\{\}\mathit{req}(0)!\{?\}.(\nu A,B)([\![P(0)]\!]_A \parallel [\![\mathit{rep}(i)!.Q]\!]_B))
\end{aligned}$$

Thus the distributed protocol deployed at nodes A and B is equal to the protocol deployed at node B alone. In other words, two hidden networks are equal if their communication capabilities are equal (proving when two recursive specifications are equal is out of scope of this paper).

7 Conclusion

We have extended Restricted Broadcast Process Theory with new operators to obtain Computed Network Theory, in which the behaviors are computed with respect to a set of topologies defined by a network restriction. Next we provided a sound and complete axiomatization of the recursion-free part of the term algebra of computed network theory, modulo the new notion of rooted branching computed network bisimilarity.

To deal with recursion, we are going to extend the axiomatization with the *Recursive Definition Principle*, the *Recursive Specification Principle*, and the *Cluster Fair Abstraction Rule* (see e.g. [5]). Applying our equational system to real-world case studies will be our next step.

References

1. Baeten, J.C.M., Bergstra, J.A., Reniers, M.A.: Discrete time process algebra with silent step. In: Proof, language, and interaction: essays in honour of Robin Milner, pp. 535–569. MIT Press, Cambridge (2000)
2. Basten, T.: Branching bisimilarity is an equivalence indeed! Inf. Process. Lett. 58(3), 141–147 (1996)
3. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control 60(1-3), 109–137 (1984)
4. Bergstra, J.A., Klop, J.W.: Algebra of communicating processes with abstraction. Theoretical Computer Science 37, 21–77 (1985)
5. Fokkink, W.J.: Introduction to Process Algebra. Springer, Heidelberg (2000)
6. Ghassemi, F., Fokkink, W.J., Movaghar, A.: Restricted broadcast process theory. In: Cerone, A., Gruner, S. (eds.) Proc. 6th Conference on Software Engineering and Formal Methods (SEFM 2008), pp. 345–354. IEEE, Los Alamitos (2008)
7. Ghassemi, F., Fokkink, W.J., Movaghar, A.: Equational reasoning on ad hoc networks. Technical report, Sharif University of Technology (2009), `http://mehr.sharif.edu/~fghassemi/Technical%20Report.pdf`
8. Godskesen, J.C.: A calculus for mobile ad hoc networks. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 132–150. Springer, Heidelberg (2007)
9. Merro, M.: An observational theory for mobile ad hoc networks. In: Proc. 23rd Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIII). Electronic Notes in Theoretical Computer Science, vol. 173, pp. 275–293. Elsevier, Amsterdam (2007)
10. Mezzetti, N., Sangiorgi, D.: Towards a calculus for wireless systems. In: Proc. 22nd Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXII). Electronic Notes in Theoretical Computer Science, vol. 158, pp. 331–353. Elsevier, Amsterdam (2006)
11. Nanz, S., Hankin, C.: A framework for security analysis of mobile wireless networks. Theoretical Computer Science 367(1), 203–227 (2006)
12. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A process calculus for mobile ad hoc networks. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 296–314. Springer, Heidelberg (2008)
13. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. Journal of the ACM 43(3), 555–600 (1996)

A Branching Computed Network Bisimilarity Is an Equivalence

To prove that branching computed network bisimilarity is an equivalence, we exploit semi-branching computed network bisimilarity, following [2]. In the next definition, $\mathcal{N} \xrightarrow{(\eta)}_C \mathcal{N}'$ denotes either $\mathcal{N} \xrightarrow{\eta}_C \mathcal{N}'$, or $\eta = m(\widehat{u})?$ and $\mathcal{N} = \mathcal{N}'$.

Definition 3. *A binary relation $\mathcal{R}$ on computed network terms is a semi-branching computed network simulation, if $\mathcal{N}_1\mathcal{R}\mathcal{N}_2$ implies whenever $\mathcal{N}_1 \xrightarrow{\eta}_C \mathcal{N}_1'$:*

- *there are $\mathcal{N}_2'$ and $\mathcal{N}_2''$ such that $\mathcal{N}_2 \Rightarrow \mathcal{N}_2'' \xrightarrow{(\eta)}_C \mathcal{N}_2'$, where $\mathcal{N}_1\mathcal{R}\mathcal{N}_2''$ and $\mathcal{N}_1'\mathcal{R}\mathcal{N}_2'$.*

$\mathcal{R}$ is a semi-branching computed network bisimulation if $\mathcal{R}$ and $\mathcal{R}^{-1}$ are semi-branching computed network simulations. Computed networks $\mathcal{N}_1$ and $\mathcal{N}_2$ are semi-branching computed network bisimilar if $\mathcal{N}_1\mathcal{R}\mathcal{N}_2$, for some semi-branching computed network bisimulation relation $\mathcal{R}$.

Lemma 1. *Let $\mathcal{N}_1$ and $\mathcal{N}_2$ be computed network terms, and $\mathcal{R}$ a semi-branching computed network bisimulation such that $\mathcal{N}_1\mathcal{R}\mathcal{N}_2$.*

- *If $\mathcal{N}_1 \Rightarrow \mathcal{N}_1'$ then $\exists \mathcal{N}_2' \cdot \mathcal{N}_2 \Rightarrow \mathcal{N}_2' \wedge \mathcal{N}_1'\mathcal{R}\mathcal{N}_2'$*
- *If $\mathcal{N}_2 \Rightarrow \mathcal{N}_2'$ then $\exists \mathcal{N}_1' \cdot \mathcal{N}_1 \Rightarrow \mathcal{N}_1' \wedge \mathcal{N}_1'\mathcal{R}\mathcal{N}_2'$*

Proof. We only give the proof of the first property. The proof is by induction on the number of $\Rightarrow$ steps from $\mathcal{N}_1$ to $\mathcal{N}_1'$:

- Base: Assume that the number of steps equals zero. Then $\mathcal{N}_1$ and $\mathcal{N}_1'$ must be equal. Since $\mathcal{N}_1\mathcal{R}\mathcal{N}_2$ and $\mathcal{N}_2 \Rightarrow \mathcal{N}_2$, the property is satisfied.
- Induction step: Assume $\mathcal{N}_1 \Rightarrow \mathcal{N}_1'$ in n steps, for some $n \geq 1$. Then there is an $\mathcal{N}_1''$ such that $\mathcal{N}_1 \Rightarrow \mathcal{N}_1''$ in $n-1 \Rightarrow$ steps, and $\mathcal{N}_1'' \xrightarrow{m(\widehat{u})?}_{\{\}} \mathcal{N}_1'$. By the induction hypothesis, there exists an $\mathcal{N}_2''$ such that $\mathcal{N}_2 \Rightarrow \mathcal{N}_2''$ and $\mathcal{N}_1''\mathcal{R}\mathcal{N}_2''$. Since $\mathcal{N}_1'' \xrightarrow{m(\widehat{u})?}_{\{\}} \mathcal{N}_1'$ and $\mathcal{R}$ is a semi-branching computed network bisimulation, there are two cases to consider:
  - there is an $\mathcal{N}_2'$ such that $\mathcal{N}_2'' \Rightarrow \mathcal{N}_2'$, $\mathcal{N}_1''\mathcal{R}\mathcal{N}_2'$, and $\mathcal{N}_1'\mathcal{R}\mathcal{N}_2'$. So $\mathcal{N}_2 \Rightarrow \mathcal{N}_2'$ such that $\mathcal{N}_1'\mathcal{R}\mathcal{N}_2'$.
  - or there are $\mathcal{N}_2'''$ and $\mathcal{N}_2'$ such that $\mathcal{N}_2'' \Rightarrow \mathcal{N}_2''' \xrightarrow{m(\widehat{u})?}_{\{\}} \mathcal{N}_2'$, where $\mathcal{N}_1''\mathcal{R}\mathcal{N}_2'''$ and $\mathcal{N}_1'\mathcal{R}\mathcal{N}_2'$. By definition, $\mathcal{N}_2''' \xrightarrow{m(\widehat{u})?}_{\{\}} \mathcal{N}_2'$ yields $\mathcal{N}_2''' \Rightarrow \mathcal{N}_2'$. Consequently $\mathcal{N}_2 \Rightarrow \mathcal{N}_2'$ such that $\mathcal{N}_1'\mathcal{R}\mathcal{N}_2'$. □

Proposition 2. *The relation composition of two semi-branching computed network bisimulations is again a semi-branching computed network bisimulation.*

Proof. Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be semi-branching computed network bisimulations with $\mathcal{N}_1\mathcal{R}_1\mathcal{N}_2$ and $\mathcal{N}_2\mathcal{R}_2\mathcal{N}_3$. Let $\mathcal{N}_1 \xrightarrow{\eta}_C \mathcal{N}_1'$. It must be shown that

$$\exists \mathcal{N}_3', \mathcal{N}_3'' : \mathcal{N}_3 \Rightarrow \mathcal{N}_3'' \xrightarrow{(\eta)} \mathcal{N}_3' \wedge \mathcal{N}_1\mathcal{R}_1 \circ \mathcal{R}_2\mathcal{N}_3'' \wedge \mathcal{N}_1'\mathcal{R}_1 \circ \mathcal{R}_2\mathcal{N}_3'$$

Since $\mathcal{N}_1\mathcal{R}_1\mathcal{N}_2$, there exist $\mathcal{N}_2'$, $\mathcal{N}_2''$ such that $\mathcal{N}_2 \Rightarrow \mathcal{N}_2'' \xrightarrow{(\eta)}_C \mathcal{N}_2'$, $\mathcal{N}_1\mathcal{R}_1\mathcal{N}_2''$ and $\mathcal{N}_1'\mathcal{R}_1\mathcal{N}_2'$. Since $\mathcal{N}_2\mathcal{R}_2\mathcal{N}_3$ and $\mathcal{N}_2 \Rightarrow \mathcal{N}_2''$, Lemma 1 yields that there is a $\mathcal{N}_3''$ such that $\mathcal{N}_3 \Rightarrow \mathcal{N}_3''$ and $\mathcal{N}_2''\mathcal{R}_2\mathcal{N}_3''$. Two cases can be distinguished:

- $\eta \in \{m(\widehat{u})?\}$ and $\mathcal{N}_2'' = \mathcal{N}_2'$. It follows immediately that $\mathcal{N}_3 \Rightarrow \mathcal{N}_3'' \xrightarrow{(\eta)}_C \mathcal{N}_3''$, $\mathcal{N}_1\mathcal{R}_1 \circ \mathcal{R}_2\mathcal{N}_3''$ and $\mathcal{N}_1'\mathcal{R}_1 \circ \mathcal{R}_2\mathcal{N}_3''$.

- Assume $\mathcal{N}_2'' \xrightarrow{\overline{\eta}}_C \mathcal{N}_2'$. Since $\mathcal{N}_2''\mathcal{R}_2\mathcal{N}_3''$ and $\mathcal{R}_2$ is a semi-branching computed network bisimulation, there are $\mathcal{N}_3'''$ and $\mathcal{N}_3'$ such that $\mathcal{N}_3'' \Rightarrow \mathcal{N}_3''' \xrightarrow{(\overline{\eta})}_C \mathcal{N}_3'$, $\mathcal{N}_2''\mathcal{R}_2\mathcal{N}_3'''$ and $\mathcal{N}_2'\mathcal{R}_2\mathcal{N}_3'$. Since $\mathcal{N}_3 \Rightarrow \mathcal{N}_3''$, we have $\mathcal{N}_3 \Rightarrow \mathcal{N}_3''' \xrightarrow{(\overline{\eta})}_C \mathcal{N}_3'$. Furthermore, $\mathcal{N}_1\mathcal{R}_1\mathcal{N}_2''\mathcal{R}_2\mathcal{N}_3'''$ and $\mathcal{N}_1'\mathcal{R}_1\mathcal{N}_2'\mathcal{R}_2\mathcal{N}_3'$. □

Corollary 3. *Semi-branching computed network bisimilarity is an equivalence relation.*

Proposition 4. *Each largest semi-branching computed network bisimulation is a branching computed network bisimulation.*

Proof. Suppose $\mathcal{R}$ is the largest semi-branching computed network bisimulation for some given constrained labeled transition systems. Let $\mathcal{N}_1\mathcal{R}\mathcal{N}_2$, $\mathcal{N}_2 \Rightarrow \mathcal{N}_2'$, $\mathcal{N}_1\mathcal{R}\mathcal{N}_2'$ and $\mathcal{N}_1'\mathcal{R}\mathcal{N}_2'$. We show that $\mathcal{R}' = \mathcal{R} \cup \{(\mathcal{N}_1', \mathcal{N}_2)\}$ is a semi-branching computed network bisimulation.

1. If $\mathcal{N}_1' \xrightarrow{\eta}_C \mathcal{N}_1''$, then it follows from $(\mathcal{N}_1', \mathcal{N}_2') \in \mathcal{R}$ that there are $\mathcal{N}_2'''$ and $\mathcal{N}_2''$ such that $\mathcal{N}_2' \Rightarrow \mathcal{N}_2''' \xrightarrow{(\overline{\eta})}_C \mathcal{N}_2''$ with $(\mathcal{N}_1', \mathcal{N}_2'''), (\mathcal{N}_1'', \mathcal{N}_2'') \in \mathcal{R}$. And $\mathcal{N}_2 \Rightarrow \mathcal{N}_2'$ yields $\mathcal{N}_2 \Rightarrow \mathcal{N}_2''' \xrightarrow{(\overline{\eta})}_C \mathcal{N}_2''$.
2. If $\mathcal{N}_2 \xrightarrow{\eta}_C \mathcal{N}_2''$, then it follows from $(\mathcal{N}_1, \mathcal{N}_2) \in \mathcal{R}$ that there are $\mathcal{N}_1'''$ and $\mathcal{N}_1''$ such that $\mathcal{N}_1 \Rightarrow \mathcal{N}_1''' \xrightarrow{(\overline{\eta})}_C \mathcal{N}_1''$ with $(\mathcal{N}_1''', \mathcal{N}_2), (\mathcal{N}_1'', \mathcal{N}_2'') \in \mathcal{R}$. Since $(\mathcal{N}_1, \mathcal{N}_2') \in \mathcal{R}$ and $\mathcal{N}_1 \Rightarrow \mathcal{N}_1'''$, by Lemma 1, there is an $\mathcal{N}_2{}'''$ such that $\mathcal{N}_2' \Rightarrow \mathcal{N}_2{}'''$ and $(\mathcal{N}_1''', \mathcal{N}_2{}''') \in \mathcal{R}$. Since $\mathcal{N}_1''' \xrightarrow{(\overline{\eta})}_C \mathcal{N}_1''$, there are $\mathcal{N}_2^{**}$ and $\mathcal{N}_2^{*}$ such that $\mathcal{N}_2{}''' \Rightarrow \mathcal{N}_2^{**} \xrightarrow{(\overline{\eta})}_C \mathcal{N}_2^{*}$ with $(\mathcal{N}_1''', \mathcal{N}_2^{**}), (\mathcal{N}_1'', \mathcal{N}_2^{*}) \in \mathcal{R}$. Since $\mathcal{N}_2' \Rightarrow \mathcal{N}_2{}'''$ and $\mathcal{N}_2{}''' \Rightarrow \mathcal{N}_2^{**}$, we have $\mathcal{N}_2' \Rightarrow \mathcal{N}_2^{**}$. By assumption, $(\mathcal{N}_1', \mathcal{N}_2') \in \mathcal{R}$, so by Lemma 1 there is an $\mathcal{N}_1^{**}$ such that $\mathcal{N}_1' \Rightarrow \mathcal{N}_1^{**}$ and $(\mathcal{N}_1^{**}, \mathcal{N}_2^{**}) \in \mathcal{R}$. Since $\mathcal{N}_2^{**} \xrightarrow{(\overline{\eta})}_C \mathcal{N}_2^{*}$, there are $\mathcal{N}_1^{***}$ and $\mathcal{N}_1^{*}$ such that $\mathcal{N}_1^{**} \Rightarrow \mathcal{N}_1^{***} \xrightarrow{(\overline{\eta})}_C \mathcal{N}_1^{*}$ with $(\mathcal{N}_1^{***}, \mathcal{N}_2^{**}), (\mathcal{N}_1^{*}, \mathcal{N}_2^{*}) \in \mathcal{R}$. And $\mathcal{N}_1' \Rightarrow \mathcal{N}_1^{**}$ yields $\mathcal{N}_1' \Rightarrow \mathcal{N}_1^{***} \xrightarrow{(\overline{\eta})}_C \mathcal{N}_1^{*}$.

$$\begin{aligned}
(\mathcal{N}_1^{***}, \mathcal{N}_2^{**}) \in \mathcal{R} \wedge (\mathcal{N}_2^{**}, \mathcal{N}_1''') \in \mathcal{R}^{-1} \wedge (\mathcal{N}_1''', \mathcal{N}_2) \in \mathcal{R} \\
\Rightarrow (\mathcal{N}_1^{***}, \mathcal{N}_2) \in \mathcal{R} \circ \mathcal{R}^{-1} \circ \mathcal{R} \\
(\mathcal{N}_1^{*}, \mathcal{N}_2^{*}) \in \mathcal{R} \wedge (\mathcal{N}_2^{*}, \mathcal{N}_1'') \in \mathcal{R}^{-1} \wedge (\mathcal{N}_1'', \mathcal{N}_2'') \in \mathcal{R} \\
\Rightarrow (\mathcal{N}_1^{*}, \mathcal{N}_2'') \in \mathcal{R} \circ \mathcal{R}^{-1} \circ \mathcal{R}
\end{aligned}$$

By Proposition 2 $\mathcal{R} \circ \mathcal{R}^{-1} \circ \mathcal{R}$ is a semi-branching computed network bisimulation. Since $\mathcal{R}$ is the largest semi-branching computed network bisimulation, and clearly $\mathcal{R} \subseteq \mathcal{R} \circ \mathcal{R}^{-1} \circ \mathcal{R}$, we have $\mathcal{R} = \mathcal{R} \circ \mathcal{R}^{-1} \circ \mathcal{R}$. Concluding, $\mathcal{N}_1' \Rightarrow \mathcal{N}_1^{***} \xrightarrow{(\overline{\eta})}_C \mathcal{N}_1^{*}$ with $(\mathcal{N}_1^{***}, \mathcal{N}_2), (\mathcal{N}_1^{*}, \mathcal{N}_2'') \in \mathcal{R}$.

So $\mathcal{R}'$ is a semi-branching computed network bisimulation. Since $\mathcal{R}$ is the largest semi-branching computed network bisimulation, $\mathcal{R}' = \mathcal{R}$.

We will now prove that $\mathcal{R}$ is a branching computed network bisimulation. Let $\mathcal{N}_1\mathcal{R}\mathcal{N}_2$, and $\mathcal{N}_1 \xrightarrow{\eta}_C \mathcal{N}_1'$. We only consider the case when η is of the form $m(\widehat{u})?$, because for other cases, the transfer condition of Definition 1 and Definition 3 are the same. So there are $\mathcal{N}_2''$ and $\mathcal{N}_2'$ such that $\mathcal{N}_2 \Rightarrow \mathcal{N}_2'' \xrightarrow{(m(\widehat{u})?)}_C \mathcal{N}_2'$ with $\mathcal{N}_1\mathcal{R}\mathcal{N}_2''$ and $\mathcal{N}_1'\mathcal{R}\mathcal{N}_2'$. Two cases can be distinguished:

1. $\mathcal{N}_2'' = \mathcal{N}_2'$: Since $\mathcal{N}_1 \mathcal{R} \mathcal{N}_2$, $\mathcal{N}_1 \mathcal{R} \mathcal{N}_2'$, and $\mathcal{N}_1' \mathcal{R} \mathcal{N}_2'$, we proved above that $\mathcal{N}_1' \mathcal{R} \mathcal{N}_2$. This agrees with the first case of Definition 1.
2. $\mathcal{N}_2'' \neq \mathcal{N}_2'$: This agrees with the second case of Definition 1.

Consequently $\mathcal{R}$ is a branching computed network bisimulation. □

Since any branching computed network bisimulation is a semi-branching computed network bisimulation, this yields the following corollary.

Corollary 5. *Two computed network terms are related by a branching computed network bisimulation if and only if they are related by a semi-branching computed network bisimulation.*

Corollary 6. *Branching computed network bisimilarity is an equivalence relation.*

Corollary 7. *Rooted branching computed network bisimilarity is an equivalence relation.*

B Rooted Branching Computed Network Bisimilarity Is a Congruence

Theorem 8. *Rooted branching computed network bisimilarity is a congruence with respect to the protocol and computed network operators.*

Proof. We need to prove that:

- $[\![P_1]\!]_\ell \simeq_{rb} [\![P_2]\!]_\ell$ implies $[\![\alpha.P_1]\!]_\ell \simeq_{rb} [\![\alpha.P_2]\!]_\ell$
- $[\![P_1]\!]_\ell \simeq_{rb} [\![P_2]\!]_\ell$ and $[\![P_1']\!]_\ell \simeq_{rb} [\![P_2']\!]_\ell$ implies $[\![P_1 + P_1']\!]_\ell \simeq_{rb} [\![P_2 + P_2']\!]_\ell$
- $[\![P_1]\!]_\ell \simeq_{rb} [\![P_2]\!]_\ell$ and $[\![P_1']\!]_\ell \simeq_{rb} [\![P_2']\!]_\ell$ implies $[\![[u_1 = u_2]P_1, P_1']\!]_\ell \simeq_{rb} [\![[u_1 = u_2]P_2, P_2']\!]_\ell$
- $\mathcal{N}_1 \simeq_{rb} \mathcal{N}_2$ implies $C\eta.\mathcal{N}_1 \simeq_{rb} C\eta.\mathcal{N}_2$
- $\mathcal{N}_1 \simeq_{rb} \mathcal{N}_2$ and $\mathcal{N}_1' \simeq_{rb} \mathcal{N}_2'$ implies $\mathcal{N}_1 + \mathcal{N}_1' \simeq_{rb} \mathcal{N}_2 + \mathcal{N}_2'$
- $\mathcal{N}_1 \simeq_{rb} \mathcal{N}_2$ implies $(\nu\ell)\mathcal{N}_1 \simeq_{rb} (\nu\ell)\mathcal{N}_2$
- $\mathcal{N}_1 \simeq_{rb} \mathcal{N}_2$ and $\mathcal{N}_1' \simeq_{rb} \mathcal{N}_2'$ implies $\mathcal{N}_1 \parallel \mathcal{N}_1' \simeq_{rb} \mathcal{N}_2 \parallel \mathcal{N}_2'$
- $\mathcal{N}_1 \simeq_{rb} \mathcal{N}_2$ and $\mathcal{N}_1' \simeq_{rb} \mathcal{N}_2'$ implies $\mathcal{N}_1 \lfloor\!\lfloor \mathcal{N}_1' \simeq_{rb} \mathcal{N}_2 \lfloor\!\lfloor \mathcal{N}_2'$
- $\mathcal{N}_1 \simeq_{rb} \mathcal{N}_2$ and $\mathcal{N}_1' \simeq_{rb} \mathcal{N}_2'$ implies $\mathcal{N}_1 \mid \mathcal{N}_1' \simeq_{rb} \mathcal{N}_2 \mid \mathcal{N}_2'$

Clearly, if $\mathcal{N}_1 \simeq_{rb} \mathcal{N}_2$ then $\mathcal{N}_1 \simeq_b \mathcal{N}_2$. Consequently the first five cases are straightforward. We prove the sixth case. To this aim we prove that if $\mathcal{N}_1 \simeq_b \mathcal{N}_2$ then $(\nu\ell)\mathcal{N}_1 \simeq_b (\nu\ell)\mathcal{N}_2$. Let $\mathcal{N}_1 \simeq_b \mathcal{N}_2$ be witnessed by the branching computed network bisimulation relation $\mathcal{R}$. We define $\mathcal{R}' = \{((\nu\ell)\mathcal{N}_1', (\nu\ell)\mathcal{N}_2') | (\mathcal{N}_1', \mathcal{N}_2') \in \mathcal{R}\}$. We prove that $\mathcal{R}'$ is a branching computed network bisimulation relation. Suppose $(\nu\ell)\mathcal{N}_1' \xrightarrow{\eta'}_{C'} (\nu\ell)\mathcal{N}_1''$ resulted from the application of $Rest$ on $\mathcal{N}_1' \xrightarrow{\eta}_C \mathcal{N}_1''$. Since $(\mathcal{N}_1', \mathcal{N}_2') \in \mathcal{R}$, there are two cases; in the first case η is a receive action and $(\mathcal{N}_1'', \mathcal{N}_2') \in \mathcal{R}$, consequently $((\nu\ell)\mathcal{N}_1'', (\nu\ell)\mathcal{N}_2') \in \mathcal{R}'$. In second case there are $\mathcal{N}_2'''$ and $\mathcal{N}_2''$ such that $\mathcal{N}_2' \Rightarrow \mathcal{N}_2''' \xrightarrow{\overline{\eta}}_C \mathcal{N}_2''$ with $(\mathcal{N}_1', \mathcal{N}_2'''), (\mathcal{N}_1'', \mathcal{N}_2'') \in \mathcal{R}$. By application of Par, $(\nu\ell)\mathcal{N}_2' \Rightarrow (\nu\ell)\mathcal{N}_2'''$ with $((\nu\ell)\mathcal{N}_1', (\nu\ell)\mathcal{N}_2''') \in \mathcal{R}'$. There are two cases to consider:

- $\overline{\eta} = \eta$: Consequently $(\nu\ell)\mathcal{N}_2''' \xrightarrow{\eta'}_{C'} (\nu\ell)\mathcal{N}_2''$.
- $\overline{\eta} \neq \eta$: in this case η is of the form $m(\widehat{u})!\{?\}$, $\eta' = \eta$ and $C' = \mathit{hide}(C, \ell)$. If $\overline{\eta} = \eta[\ell/?]$ then $\overline{\eta}[?/\ell] = \eta$ and $C' = \mathit{hide}(C[\ell/?], \ell)$ hold, otherwise $\overline{\eta}[?/\ell] = \overline{\eta}$ and $C'[\ell'/?] = \mathit{hide}(C[\ell'/?], \ell)$ hold where $\ell' \neq \ell$. Consequently $(\nu\ell)\mathcal{N}_2''' \xrightarrow{\overline{\eta'}}_{C'} (\nu\ell)\mathcal{N}_2''$.

With the above argumentation, there are $\mathcal{N}_2'''$ and $\mathcal{N}_2''$ such that $(\nu\ell)\mathcal{N}_2' \Rightarrow (\nu\ell)\mathcal{N}_2''' \xrightarrow{\overline{\eta'}}_{C'} (\nu\ell)\mathcal{N}_2''$ with $((\nu\ell)\mathcal{N}_1', (\nu\ell)\mathcal{N}_2'''), ((\nu\ell)\mathcal{N}_1'', (\nu\ell)\mathcal{N}_2'') \in \mathcal{R}'$.

Likewise we can prove that $\mathcal{N}_1 \simeq_{rb} \mathcal{N}_2$ implies $(\nu\ell)\mathcal{N}_1 \simeq_{rb} (\nu\ell)\mathcal{N}_2$. To this aim we examine the root condition in Definition 2. Suppose $(\nu\ell)\mathcal{N}_1 \xrightarrow{\eta'}_{C'} (\nu\ell)\mathcal{N}_1'$, with the same argumentation as above, $(\nu\ell)\mathcal{N}_2 \xrightarrow{\overline{\eta'}}_{C'} (\nu\ell)\mathcal{N}_2'$. Since $\mathcal{N}_1' \simeq_b \mathcal{N}_2'$, we proved that $(\nu\ell)\mathcal{N}_1' \simeq_b (\nu\ell)\mathcal{N}_2'$. Concluding $(\nu\ell)\mathcal{N}_1 \simeq_{rb} (\nu\ell)\mathcal{N}_2$.

From the three remaining cases, we focus on the most challenging case, which is the sync operator $|$; the others are proved in a similar fashion. First we prove that if $\mathcal{N}_1 \simeq_b \mathcal{N}_2$, then $\mathcal{N}_1 \parallel \mathcal{N} \simeq_b \mathcal{N}_2 \parallel \mathcal{N}$. Let $\mathcal{N}_1 \simeq_b \mathcal{N}_2$ be witnessed by the branching computed network bisimulation relation $\mathcal{R}$. We define $\mathcal{R}' = \{(\mathcal{N}_1' \parallel \mathcal{N}', \mathcal{N}_2' \parallel \mathcal{N}') | (\mathcal{N}_1', \mathcal{N}_2') \in \mathcal{R}, \mathcal{N}'$ any computed network term$\}$. We prove that $\mathcal{R}'$ is a branching computed network bisimulation relation. Suppose $\mathcal{N}_1 \parallel \mathcal{N} \xrightarrow{\eta}_{C^*} \mathcal{N}^*$. There are several cases to consider:

- Suppose η is a send action $m(\widehat{u})!$ performed by an address ℓ. First let it be performed by $\mathcal{N}_1'$, and $\mathcal{N}$ participated in the communication. That is, $\mathcal{N}_1' \xrightarrow{m(\widehat{u})!\{\ell\}}_{C_1} \mathcal{N}_1''$ and $\mathcal{N} \xrightarrow{m(\widehat{u})?}_{C} \mathcal{N}'$ give rise to the transition $\mathcal{N}_1' \parallel \mathcal{N} \xrightarrow{m(\widehat{u})!\{\ell\}}_{C_1 \cup C[\ell/?]} \mathcal{N}_1'' \parallel \mathcal{N}'$. As $(\mathcal{N}_1', \mathcal{N}_2') \in \mathcal{R}$ and $\mathcal{N}_1' \xrightarrow{m(\widehat{u})!\{\ell\}}_{C_1} \mathcal{N}_1''$, there are $\mathcal{N}_2''$ and $\mathcal{N}_2'''$ such that $\mathcal{N}_2' \Rightarrow \mathcal{N}_2''' \xrightarrow{m(\widehat{u})!\{\ell'\}}_{C_1[\ell'/\ell]} \mathcal{N}_2''$, where $(\ell = ? \vee \ell = \ell')$ and $(\mathcal{N}_1', \mathcal{N}_2'''), (\mathcal{N}_1'', \mathcal{N}_2'') \in \mathcal{R}$. Hence $\mathcal{N}_2' \parallel \mathcal{N} \Rightarrow \mathcal{N}_2''' \parallel \mathcal{N} \xrightarrow{m(\widehat{u})!\{\ell'\}}_{C_1 \cup C[\ell'/?]} \mathcal{N}_2'' \parallel \mathcal{N}'$ with $(\mathcal{N}_1' \parallel \mathcal{N}, \mathcal{N}_2''' \parallel \mathcal{N}), (\mathcal{N}_1'' \parallel \mathcal{N}', \mathcal{N}_2'' \parallel \mathcal{N}') \in \mathcal{R}'$.
  Now suppose that the send action was performed by $\mathcal{N}$, and $\mathcal{N}_1'$ participated in the communication. That is, $\mathcal{N}_1' \xrightarrow{m(\widehat{u})?}_{C_1} \mathcal{N}_1''$ and $\mathcal{N} \xrightarrow{m(\widehat{u})!\{\ell\}}_{C} \mathcal{N}'$ give rise to the transition $\mathcal{N}_1' \parallel \mathcal{N} \xrightarrow{m(\widehat{u})!\{\ell\}}_{C_1[\ell/?] \cup C} \mathcal{N}_1'' \parallel \mathcal{N}'$. Since $(\mathcal{N}_1', \mathcal{N}_2') \in \mathcal{R}$ and $\mathcal{N}_1' \xrightarrow{m(\widehat{u})?}_{C_1} \mathcal{N}_1''$, two cases can be considered: either $(\mathcal{N}_1'', \mathcal{N}_2') \in \mathcal{R}$, or there are $\mathcal{N}_2'''$ and $\mathcal{N}_2''$ such that $\mathcal{N}_2' \Rightarrow \mathcal{N}_2''' \xrightarrow{m(\widehat{u})?}_{C_1} \mathcal{N}_2''$ with $(\mathcal{N}_1', \mathcal{N}_2'''), (\mathcal{N}_1'', \mathcal{N}_2'') \in \mathcal{R}$. In the first case, $\mathcal{N}_2' \parallel \mathcal{N} \xrightarrow{m(\widehat{u})!\{\ell\}}_{C_1 \cup C[\ell/?]} \mathcal{N}_2' \parallel \mathcal{N}'$, and $(\mathcal{N}_1'' \parallel \mathcal{N}', \mathcal{N}_2' \parallel \mathcal{N}') \in \mathcal{R}$. In the second case, $\mathcal{N}_2' \parallel \mathcal{N} \Rightarrow \mathcal{N}_2''' \parallel \mathcal{N} \xrightarrow{m(\widehat{u})!\{\ell\}}_{C_1 \cup C[\ell/?]} \mathcal{N}_2'' \parallel \mathcal{N}'$, and $(\mathcal{N}_1' \parallel \mathcal{N}, \mathcal{N}_2''' \parallel \mathcal{N}), (\mathcal{N}_1'' \parallel \mathcal{N}', \mathcal{N}_2'' \parallel \mathcal{N}') \in \mathcal{R}'$.
  The cases where $\mathcal{N}$ or $\mathcal{N}_1$ does not participate in the communication are straightforward.
- The case where η is a receive action $m(\widehat{u})?$ is also straightforward; it originates from $\mathcal{N}_1, \mathcal{N}$, or both.

Likewise we can prove that $\mathcal{N}_1 \simeq_{rb} \mathcal{N}_2$ implies $\mathcal{N} \parallel \mathcal{N}_1 \simeq_{rb} \mathcal{N} \parallel \mathcal{N}_2$.

Now let $\mathcal{N}_1 \simeq_{rb} \mathcal{N}_2$. To prove $\mathcal{N}_1|\mathcal{N} \simeq_{rb} \mathcal{N}_2|\mathcal{N}$, we examine the root condition from Definition 2. Suppose $\mathcal{N}_1|\mathcal{N} \xrightarrow{m(\widehat{u})!\{\ell\}}_{C^*} \mathcal{N}^*$. There are two cases to consider:

- This send action was performed by $\mathcal{N}_1$ at node ℓ, and $\mathcal{N}$ participated in the communication. That is, $\mathcal{N}_1 \xrightarrow{m(\widehat{u})!\{\ell\}}_{C_1} \mathcal{N}_1'$ and $\mathcal{N} \xrightarrow{m(\widehat{u})?}_{C} \mathcal{N}'$, so that $\mathcal{N}_1|\mathcal{N} \xrightarrow{m(\widehat{u})!\{\ell\}}_{C_1 \cup C[\ell/?]} \mathcal{N}_1' \parallel \mathcal{N}'$. Since $\mathcal{N}_1 \simeq_{rb} \mathcal{N}_2$, there is an $\mathcal{N}_2'$ such that $\mathcal{N}_2 \xrightarrow{m(\widehat{u})!\{\ell'\}}_{C_1[\ell'/\ell]} \mathcal{N}_2'$ with $(\ell = ? \vee \ell = \ell')$ and $\mathcal{N}_1' \simeq_b \mathcal{N}_2'$. Then $\mathcal{N}_2|\mathcal{N} \xrightarrow{m(\widehat{u})!\{\ell'\}}_{C_1 \cup C[\ell'/?]} \mathcal{N}_2' \parallel \mathcal{N}'$. Since $\mathcal{N}_1' \simeq_b \mathcal{N}_2'$, we proved that $\mathcal{N}_1' \parallel \mathcal{N}' \simeq_b \mathcal{N}_2' \parallel \mathcal{N}'$.
- The send action was performed $\mathcal{N}$ at node ℓ, and $\mathcal{N}_1$ participated in the communication. That is, $\mathcal{N}_1 \xrightarrow{m(\widehat{u})?}_{C_1} \mathcal{N}_1'$ and $\mathcal{N} \xrightarrow{m(\widehat{u})!\{\ell\}}_{C} \mathcal{N}$, so that $\mathcal{N}_1|\mathcal{N} \xrightarrow{m(\widehat{u})!\{\ell\}}_{C_1 \cup C[\ell/?]} \mathcal{N}_1 \parallel \mathcal{N}'$. Since $\mathcal{N}_1 \simeq_{rb} \mathcal{N}_2$, there is an $\mathcal{N}_2'$ such that $\mathcal{N}_2 \xrightarrow{m(\widehat{u})?}_{C_1} \mathcal{N}_2'$ with $\mathcal{N}_1' \simeq_b \mathcal{N}_2'$. Then $\mathcal{N}_2|\mathcal{N} \xrightarrow{m(\widehat{u})!\{\ell\}}_{C_1 \cup C[\ell/?]} \mathcal{N}_2' \parallel \mathcal{N}'$. Since $\mathcal{N}_1' \simeq_b \mathcal{N}_2'$, we have $\mathcal{N}_1' \parallel \mathcal{N}' \simeq_b \mathcal{N}_2' \parallel \mathcal{N}'$.

Finally, the case where $\mathcal{N}_1|\mathcal{N} \xrightarrow{m(\widehat{u})?}_{C^*} \mathcal{N}^*$ can be easily dealt with. This receive action was performed by both $\mathcal{N}_1$ and $\mathcal{N}$.

Concluding, $\mathcal{N}_1|\mathcal{N} \simeq_{rb} \mathcal{N}_2|\mathcal{N}$. Likewise it can be argued that $\mathcal{N}|\mathcal{N}_1 \simeq_{rb} \mathcal{N}|\mathcal{N}_2$. □

Towards a Notion of Unsatisfiable Cores for LTL

Viktor Schuppan

FBK-irst, Via Sommarive 18, 38123 Trento, Italy
schuppan@fbk.eu

Abstract. Unsatisfiable cores, i.e., parts of an unsatisfiable formula that are themselves unsatisfiable, have important uses in debugging specifications, speeding up search in model checking or SMT, and generating certificates of unsatisfiability. While unsatisfiable cores have been well investigated for Boolean SAT and constraint programming, the notion of unsatisfiable cores for temporal logics such as LTL has not received much attention. In this paper we investigate notions of unsatisfiable cores for LTL that arise from the syntax tree of an LTL formula, from converting it into a conjunctive normal form, and from proofs of its unsatisfiability. The resulting notions are more fine-granular than existing ones.

1 Introduction

Temporal logics such as LTL have become a standard formalism to specify requirements for reactive systems [37]. Hence, in recent years methodologies for property-based design based on temporal logics have been developed (e.g., [1]).

Increasing use of temporal logic requirements in the design process necessitates the availability of efficient validation and debugging methodologies. Vacuity checking [5, 31] and coverage [12] are complementary approaches developed in the context of model checking (e.g., [3]) for validating requirements given as temporal logic properties. However, with the exception of [13, 24], both vacuity and coverage assume presence of both a model and its requirements. Particularly in early stages of the design process the former might not be available. Satisfiability and realizability [38] checking are approaches that can handle requirements without a model being avaiable. Tool support for both is available (e.g., [8]).

Typically, unsatisfiability of a set of requirements signals presence of a problem; finding a reason for unsatisfiability can help with the ensuing debugging. In practice, determining a reason for unsatisfiability of a formula without automated support is often doomed to fail due to the sheer size of the formula. Consider, e.g., the EURAILCHECK project that developed a methodology and a tool for the validation of requirements [18]. Part of the methodology consists of translating the set of requirements given by a textual specification into a variant of LTL and subsequent checking for satisfiability; if the requirements are unsatisfiable, an unsatisfiable subset of them is returned to the user. The textual specification considered as a feasibility study is a few 100 pages long.

Another application for determining reasons for unsatisfiability are algorithms that find a solution to a problem in an iterative fashion. They start with a guess of a solution and check whether that guess is indeed a solution. If not, rather

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 129–145, 2010.

than ruling out only that guess, they determine a reason for that guess not being a solution and rule out guesses doomed to fail for the same reason. Examples are verification with CEGAR (e.g., [20]) and SMT (e.g., [47]). Automated support for determining a reason for unsatisfiability is clearly essential.

Current implementations for satisfiability checking (e.g., [16]) point out reasons for unsatisfiability by returning a part of an unsatisfiable formula that is by itself unsatisfiable. This is called an unsatisfiable core (UC). However, these UCs are coarse-grained in the following sense. The input formula is a Boolean combination of temporal logic formulas. When extracting an UC current implementations do not look inside temporal subformulas: when, e.g., $\phi = (\mathbf{G}\psi) \wedge (\mathbf{F}\psi')$ is found to be unsatisfiable, then [16] will return ϕ as an UC irrespective of the complexity of ψ and ψ'. Whether the resulting core is inspected for debugging by a human or used as a filter in a search process by a machine: in either case a more fine-granular UC will likely make the corresponding task easier.

In this paper we take first steps to overcome the restrictions of UCs for LTL by investigating more fine-grained notions of UCs for LTL. We start with a notion based on the syntactic structure of the input formula where entire subformulas are replaced with 1 (true) or 0 (false) depending on the polarity of the corresponding subformula. We then consider conjunctive normal forms obtained by structure-preserving clause form translations [36]; the resulting notion of core is one of a subset of conjuncts. That notion is reused when looking at UCs extracted from resolution proofs from bounded model checking (BMC) [6] runs. We finally show how to extract an UC from a tableau proof [25] of unsatisfiability. All 4 notions can express UCs that are as fine-grained as the one based on the syntactic formula structure. The notion based on conjunctive normal forms provides more fine-grained resolution in the temporal dimension, and those based on BMC and on unsatisfied tableau proofs raise the hope to do even better. At this point we would like to emphasize the distinction between notions of UCs and methods to obtain them. While there is some emphasis in this paper on methods for UC extraction, here we see such methods only as a vehicle to suggest notions of UCs. We are not aware of similar systematic investigation of the notion of UC for LTL; for notions of cores for other formalisms, for application of UCs, and for technically related approaches such as vacuity checking see Sect. 8.

In the next Sect. 2 we state the preliminaries and in Sect. 3 we introduce some general notions. In Sect.s 4, 5, 6, and 7 we investigate UCs obtained by syntactic manipulation of parse trees, by taking subsets of conjuncts in conjunctive normal forms, by extracting resolution proofs from BMC runs, and by extraction from closed tableaux nodes. Related work is discussed in Sect. 8 before we conclude in Sect. 9. We do not provide a formalization of some parts and discussion of some aspects in this extended abstract but instead refer to the full version [41].

2 Preliminaries

In the following we give standard definitions for LTL, see, e.g., [3]. Let $\mathbb{B}$ be the set of Booleans, $\mathbb{N}$ the naturals, and AP a finite set of atomic propositions.

Definition 1 (LTL Syntax). *The set of* LTL formulas *is constructed inductively as follows. Boolean constants* $0, 1 \in \mathbb{B}$ *and atomic propositions* $p \in AP$ *are LTL formulas. If* ψ, ψ' *are LTL formulas, so are* $\neg\psi$, $\psi \vee \psi'$, $\psi \wedge \psi'$, $\mathbf{X}\psi$, $\psi\mathbf{U}\psi'$, $\psi\mathbf{R}\psi'$, $\mathbf{F}\psi$, *and* $\mathbf{G}\psi$. *We use* $\psi \rightarrow \psi'$ *as an abbreviation for* $\neg\psi \vee \psi'$, $\psi \leftarrow \psi'$ *for* $\psi \vee \neg\psi'$, *and* $\psi \leftrightarrow \psi'$ *for* $(\psi \rightarrow \psi') \wedge (\psi \leftarrow \psi')$.

The semantics of LTL formulas is defined on infinite words over the alphabet 2^{AP}. If π is an infinite word in $(2^{AP})^\omega$ and i is a position in $\mathbb{N}$, then $\pi[i]$ denotes the letter at the i-th position of π and $\pi[i, \infty]$ denotes the suffix of π starting at position i (inclusive). We now inductively define the semantics of an LTL formula on positions $i \in \mathbb{N}$ of a word $\pi \in (2^{AP})^\omega$:

Definition 2 (LTL Semantics).

$$\begin{array}{lll}
(\pi,i) \models 1 & (\pi,i) \not\models 0 & (\pi,i) \models \psi\mathbf{U}\psi' \Leftrightarrow \exists i' \geq i \,.\, ((\pi,i') \models \psi' \wedge \forall i \leq i'' < i' \,.\, (\pi,i'') \models \psi) \\
(\pi,i) \models p & \Leftrightarrow p \in \pi[i] & (\pi,i) \models \psi\mathbf{R}\psi' \Leftrightarrow \forall i' \geq i \,.\, ((\pi,i') \models \psi' \vee \exists i \leq i'' < i' \,.\, (\pi,i'') \models \psi) \\
(\pi,i) \models \neg\psi & \Leftrightarrow (\pi,i) \not\models \psi & \qquad\qquad (\pi,i) \models \mathbf{X}\psi \Leftrightarrow (\pi,i+1) \models \psi \\
(\pi,i) \models \psi \vee \psi' & \Leftrightarrow (\pi,i) \models \psi \textit{ or } (\pi,i) \models \psi' & \qquad\qquad (\pi,i) \models \mathbf{F}\psi \Leftrightarrow \exists i' \geq i \,.\, (\pi,i') \models \psi \\
(\pi,i) \models \psi \wedge \psi' & \Leftrightarrow (\pi,i) \models \psi \textit{ and } (\pi,i) \models \psi' & \qquad\qquad (\pi,i) \models \mathbf{G}\psi \Leftrightarrow \forall i' \geq i \,.\, (\pi,i') \models \psi
\end{array}$$

An infinite word π satisfies *a formula* ϕ *iff the formula holds at the beginning of that word:* $\pi \models \phi \Leftrightarrow (\pi, 0) \models \phi$. *Then we call* π *a satisfying assignment to* ϕ.

Definition 3 (Satisfiability). *An LTL formula* ϕ *is* satisfiable *if there exists a word* π *that satisfies it:* $\exists \pi \in (2^{AP})^\omega \,.\, \pi \models \phi$; *it is* unsatisfiable *otherwise.*

Definition 4 (Negation Normal Form). *An LTL formula* ϕ *is in* negation normal form (NNF) $nnf(\phi)$ *if negations are applied only to atomic propositions.*

Definition 5 (Subformula). *Let* ϕ *be an LTL formula. The set of subformulas* $SF(\phi)$ *of* ϕ *is defined recursively as follows:*

$$\begin{array}{llll}
\psi = b \textit{ or } \psi = p & \textit{with} & b \in \mathbb{B}, p \in AP & : \; SF(\psi) = \{\psi\} \\
\psi = \circ_1 \psi' & \textit{with} & \circ_1 \in \{\neg, \mathbf{X}, \mathbf{F}, \mathbf{G}\} & : \; SF(\psi) = \{\psi\} \cup SF(\psi') \\
\psi = \psi' \circ_2 \psi'' & \textit{with} & \circ_2 \in \{\vee, \wedge, \mathbf{U}, \mathbf{R}\} & : \; SF(\psi) = \{\psi\} \cup SF(\psi') \cup SF(\psi'')
\end{array}$$

Definition 6 (Polarity). *Let* ϕ *be an LTL formula, let* $\psi \in SF(\phi)$. ψ *has* positive polarity *(+) in* ϕ *if it appears under an even number of negations,* negative polarity *(−) otherwise.*

We regard LTL formulas as trees, i.e., we don't take sharing of subformulas into account. We don't attempt to simplify formulas before or after UC extraction.

3 Notions and Concepts Related to UCs

In this section we discuss general notions in the context of UCs[1] independently of the notion of UC used. It is not a goal of this paper to formalize the notions below towards a general framework of UCs. Instead, in the remainder of this paper we focus on the case of LTL where instantiations are readily available.

[1] Terminology in the literature for these notions is diverse. We settled for the term "unsatisfiable core", which is used for such notions, e.g., in the context of Boolean satisfiability (e.g., [26,48]), SMT (e.g., [14]), and declarative specifications (e.g., [45]).

UCs, Irreducible UCs, and Least-Cost Irreducible UCs When dealing with *UCs* one typically considers an input ϕ (here: LTL formula) taken from a set of possible inputs Φ (here: all LTL formulas) and a Boolean-valued function $\mathit{foo}^2 : \Phi \mapsto \mathbb{B}$ with $\mathit{foo}(\phi) = 0$ (here: LTL satisfiability). The goal is to derive another input ϕ' (the UC) with $\mathit{foo}(\phi') = 0$ from ϕ s.t. 1. the derivation preserves a sufficient set of reasons for *foo* being 0 without adding new reasons, 2. the fact that $\mathit{foo}(\phi')$ is 0 is easier to see for the user than the fact that $\mathit{foo}(\phi)$ is 0, and 3. the derivation is such that preservance/non-addition of reasons for *foo* being 0 on ϕ and ϕ' can be understood by the user. Typically 1 and 3 are met by limiting the derivation to some set of operations on inputs that fulfills these criteria (here: syntactic weakening of LTL formulas). The remaining criterion 2 can be handled by assuming a cost function on inputs where lower cost provides some reason to hope for easier comprehension by the user (here: see below).

Assuming a set of inputs and a set of operations we can define the following notions. An input ϕ' is called a *core* of an input ϕ if it is derived by a sequence of such operations. ϕ' is an *unsatisfiable* core if ϕ' is a core of ϕ and $\mathit{foo}(\phi') = 0$. ϕ' is a *proper* unsatisfiable core if ϕ' is an unsatisfiable core of ϕ and is syntactically different from ϕ. Finally, ϕ' is an *irreducible* unsatisfiable core (IUC) if ϕ' is an unsatisfiable core of ϕ and there is no proper unsatisfiable core of ϕ'. Often IUCs are called minimal UCs and least-cost IUCs minimum UCs.

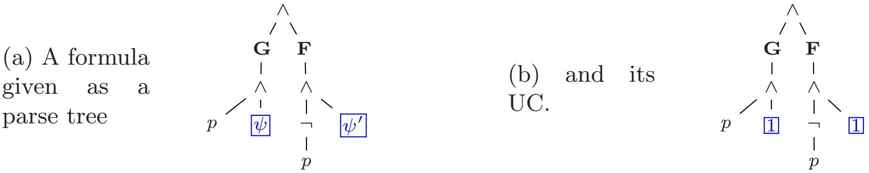
Cost functions often refer to some size measure of an input as suggested by a specific notion of core, e.g., the number of conjuncts when inputs are conjunctions of formulas and *foo* is satisfiability. We do not consider specific cost functions.

Granularity of a Notion of UC. Clearly, the original input contains at least as much information as any of its UCs and, in particular, all reasons for being unsatisfiable. However, our goal when defining notions of UCs is to come up with derived inputs that make some of these reasons easier to see. Therefore we use the term *granularity* of a notion of core as follows. We wish to determine the relevance of certain aspects of an input to the input being unsatisfiable by the mere presence or absence of elements in the UC. In other words, we do not take potential steps of inference by the user into account. Hence, we say that one notion of core provides finer granularity than another if it provides at least as much information on the relevance of certain aspects of an input as the other. Consider, e.g., a notion of UC that takes a set of formulas as input and defines a core to be a subset of this set without proceeding to modify the member formulas versus a notion that also modifies the member formulas. Another example is a notion of UC for LTL that considers relevance of subformulas at certain points in time versus a notion that only either keeps or discards subformulas.

4 Unsatisfiable Cores via Parse Trees

In this section we consider UCs purely based on the syntactic structure of the formula. It is easy to see that, as is done, e.g., in some forms of vacuity checking

[2] Although we write *foo* we still say “unsatisfiable” core rather than “unfooable” core.


(a) A formula given as a parse tree

(b) and its UC.

Fig. 1. Example of an UC via parse tree. Modified parts are marked blue boxed.

[31], replacing an occurrence of a subformula with positive polarity with 1 or replacing an occurrence of a subformula with negative polarity with 0 will lead to a weaker formula. This naturally leads to a definition of UC based on parse trees where replacing occurrences of subformulas corresponds to replacing subtrees.

Consider the following formula $\phi = (\mathbf{G}(p \wedge \psi)) \wedge (\mathbf{F}(\neg p \wedge \psi'))$ whose parse tree is depicted in Fig. 1 (a). The formula is unsatisfiable independent of the concrete (and possibly complex) subformulas ψ, ψ'. A corresponding UC with ψ, ψ' replaced with 1 is $\phi' = (\mathbf{G}(p \wedge 1)) \wedge (\mathbf{F}(\neg p \wedge 1))$, shown in Fig. 1 (b).

Hence, by letting the set of operations to derive a core be replacement of occurrences of subformulas of ϕ with 1 (for positive polarity occurrences) or 0 (for negative polarity occurrences), we obtain the notions of core, unsatisfiable core, proper unsatisfiable core, and irreducible unsatisfiable core *via parse tree*.

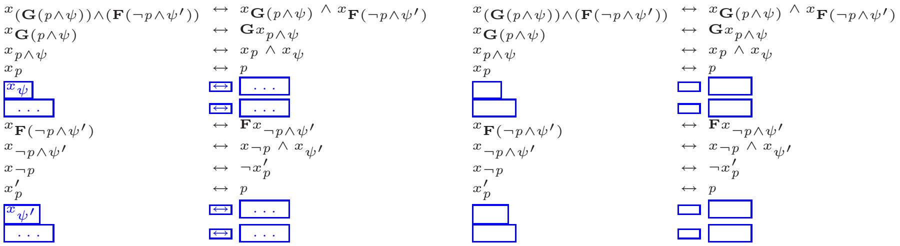
In the example above ϕ' is both a proper and an IUC of ϕ. Note that $(\mathbf{G}(p \wedge 1)) \wedge (\mathbf{F}(\neg p \wedge \psi'))$ and $(\mathbf{G}(p \wedge \psi)) \wedge (\mathbf{F}(\neg p \wedge 1))$ are UCs of ϕ, too, as is ϕ itself (and possibly many more when ψ and ψ' are taken into account).

5 Unsatisfiable Cores via Definitional Conjunctive Normal Form

Structure preserving translations (e.g., [36]) of formulas into conjunctive normal form introduce fresh Boolean propositions for (some) subformulas that are constrained by one or more conjuncts to be 1 (if and) only if the corresponding subformulas hold in some satisfying assignment. In this paper we use the term definitional conjunctive normal form (dCNF) to make a clear distinction from the conjunctive normal form used in Boolean satisfiability (SAT), which we denote CNF. dCNF is often a preferred representation of formulas as it's typically easy to convert a formula into dCNF, the expansion in formula size is moderate, and the result is frequently amenable to resolution. Most important in the context of this paper, dCNFs yield a straightforward and most commonly used notion of core in the form of a (possibly constrained) subset of conjuncts.

5.1 Basic Form

Below we define the basic version of dCNF. It is well-known that ϕ and $\mathit{dCNF}(\phi)$ are equisatisfiable.


$x_{(\mathbf{G}(p\wedge\psi))\wedge(\mathbf{F}(\neg p\wedge\psi'))} \leftrightarrow x_{\mathbf{G}(p\wedge\psi)} \wedge x_{\mathbf{F}(\neg p\wedge\psi')}$
$x_{\mathbf{G}(p\wedge\psi)} \leftrightarrow \mathbf{G}x_{p\wedge\psi}$
$x_{p\wedge\psi} \leftrightarrow x_p \wedge x_\psi$
$x_p \leftrightarrow p$
$x_\psi \leftrightarrow \ldots$
$\ldots \leftrightarrow \ldots$
$x_{\mathbf{F}(\neg p\wedge\psi')} \leftrightarrow \mathbf{F}x_{\neg p\wedge\psi'}$
$x_{\neg p\wedge\psi'} \leftrightarrow x_{\neg p} \wedge x_{\psi'}$
$x_{\neg p} \leftrightarrow \neg x'_p$
$x'_p \leftrightarrow p$
$x_{\psi'} \leftrightarrow \ldots$
$\ldots \leftrightarrow \ldots$

(a) A formula given as a dCNF (b) and its UC.

Fig. 2. Example of UC via dCNF for $\phi = (\mathbf{G}(p \wedge \psi)) \wedge (\mathbf{F}(\neg p \wedge \psi'))$. The "..." stand for definitions of ψ, ψ', and their subformulas. Modified parts are marked blue boxed.

Definition 7 (Definitional Conjunctive Normal Form). *Let ϕ be an LTL formula over atomic propositions AP, let $x, x', \ldots \in X$ be fresh atomic propositions not in AP. $dCNF_{aux}(\phi)$ is a set of conjuncts containing one conjunct for each occurrence of a subformula ψ in ϕ as follows:*

ψ	*Conjunct* $\in dCNF_{aux}(\phi)$	ψ	*Conjunct* $\in dCNF_{aux}(\phi)$
$b \in \mathbb{B}$	$x_\psi \leftrightarrow b$	$\circ_1 \psi'$ *with* $\circ_1 \in \{\neg, \mathbf{X}, \mathbf{F}, \mathbf{G}\}$	$x_\psi \leftrightarrow \circ_1 x_{\psi'}$
$p \in AP$	$x_\psi \leftrightarrow p$	$\psi' \circ_2 \psi''$ *with* $\circ_2 \in \{\vee, \wedge, \mathbf{U}, \mathbf{R}\}$	$x_\psi \leftrightarrow x_{\psi'} \circ_2 x_{\psi''}$

Then the definitional conjunctive normal form *of ϕ is defined as*

$$dCNF(\phi) \equiv x_\phi \wedge \mathbf{G} \textstyle\bigwedge_{c \in dCNF_{aux}(\phi)} c$$

x_ϕ is called the root *of the dCNF. An occurrence of x on the left-hand side of a biimplication is a* definition *of x, an occurrence on the right-hand side a* use.

By letting the operations to derive a core from an input be the removal of elements of $dCNF_{aux}(\phi)$ we obtain the notions of core, unsatisfiable core, proper unsatisfiable core, and irreducible unsatisfiable core *via dCNF*. We additionally require that all conjuncts are discarded that contain definitions for which no (more) conjunct with a corresponding use exists.

We continue the example from Fig. 1 in Fig. 2. In the figure we identify an UC with its set of conjuncts. In Fig. 2 (b) the definitions for both ψ and ψ' and all dependent definitions are removed. As in Sect. 4 the UC shown in Fig. 2 (b) is an IUC with more UCs existing.

Correspondence Between Cores via Parse Trees and via dCNF. Let ϕ be an LTL formula. From Def. 7 it is clear that there is a one-to-one correspondence between the nodes in the parse tree of ϕ and the conjuncts in its dCNF. Therefore, the conversion between the representation of ϕ as a parse tree and as a dCNF is straightforward. Remember that an UC of a parse tree is obtained by replacing an occurrence of a subformula ψ with 1 or 0, while an UC of a dCNF is obtained by removing the definition of ψ and all dependent definitions. Both ways to obtain an UC do not destroy the correspondence between parse trees and dCNFs. Hence, the notions of UC obtained via parse tree and via dCNF are equivalent.

5.2 Variants

We now examine some variants of Def. 7 w.r.t. the information contained in the UCs that they can yield. Each variant is built on top of the previous one.

Replacing Biimplications with Implications. Definition 7 uses biimplication rather than implication in order to cover the case of both positive and negative polarity occurrences of subformulas in a uniform way. A seemingly refined variant is to consider both directions of that biimplication separately.[3] However, it is easy to see that in our setting of formulas as parse trees, i.e., without sharing of subformulas, each subformula has a unique polarity and, hence, only one direction of the biimplication will be present in an IUC. I.o.w., using 2 implications rather than a biimplication has no benefit in terms of granularity of the obtained cores.

Splitting Implications for Binary Operators. We now consider left-hand and right-hand operands of the $\wedge$ and $\vee$ operators separately by splitting the implications for $\wedge$ and the reverse implications for $\vee$ into two. For example, $x_{\psi' \wedge \psi''} \rightarrow x_{\psi'} \wedge x_{\psi''}$ is split into $x_{\psi' \wedge \psi''} \rightarrow x_{\psi'}$ and $x_{\psi' \wedge \psi''} \rightarrow x_{\psi''}$. That variant can be seen not to yield finer granularity as follows. Assume an IUC *dCNF'* contains a conjunct $x_{\psi' \wedge \psi''} \rightarrow x_{\psi'}$ but not $x_{\psi' \wedge \psi''} \rightarrow x_{\psi''}$. The corresponding IUC *dCNF* based on Def. 7 must contain the conjunct $x_{\psi' \wedge \psi''} \rightarrow x_{\psi'} \wedge x_{\psi''}$ but will not contain a definition of $x_{\psi''}$. Hence, also in the IUC based on Def. 7, the subformula occurrence ψ'' can be seen to be irrelevant to that core. The case for $\vee$ is similar.

Temporal Unfolding. Here we rewrite a conjunct for a positive polarity occurrence of an $\mathbf{U}$ subformula as its one-step temporal unfolding and an additional conjunct to enforce the desired fixed point. I.e., we replace a conjunct $x_{\psi' \mathbf{U} \psi''} \rightarrow x_{\psi'} \mathbf{U} x_{\psi''}$ with $x_{\psi' \mathbf{U} \psi''} \rightarrow x_{\psi''} \vee (x_{\psi'} \wedge \mathbf{X} x_{\psi' \mathbf{U} \psi''})$ and $x_{\psi' \mathbf{U} \psi''} \rightarrow \mathbf{F} x_{\psi''}$.

This can be seen to provide improved information for positive polarity occurrences of $\mathbf{U}$ subformulas in an IUC as follows. A dCNF for a positive occurrence of an $\mathbf{U}$ subformula $\psi' \mathbf{U} \psi''$ obtained without temporal unfolding as in the previous variant results (among others) in the following conjuncts: $c = x_{\psi' \mathbf{U} \psi''} \rightarrow x_{\psi'} \mathbf{U} x_{\psi''}$, $C''' = \{x_{\psi'} \rightarrow \ldots\}$, and $C'''' = \{x_{\psi''} \rightarrow \ldots\}$. An IUC based on that dCNF contains either 1. none of c, $c''' \in C'''$, $c'''' \in C''''$, 2. c, $c'''' \in C''''$, or 3. c, $c''' \in C'''$, $c'''' \in C''''$. O.t.o.h., a dCNF with temporal unfolding results in the conjuncts: $c' = x_{\psi' \mathbf{U} \psi''} \rightarrow x_{\psi''} \vee (x_{\psi'} \wedge \mathbf{X} x_{\psi' \mathbf{U} \psi''})$, $c'' = x_{\psi' \mathbf{U} \psi''} \rightarrow \mathbf{F} x_{\psi''}$, and C''', C'''' as before. An IUC based on that dCNF contains either 1. none of c', c'', $c''' \in C'''$, $c'''' \in C''''$, 2. c', $c''' \in C'''$, $c'''' \in C''' \in C''''$, 3. c'', $c'''' \in C''''$, or 4. c', c'', $c''' \in C'''$, $c'''' \in C''''$. For some $\mathbf{U}$ subformulas the additional case allows to distinguish between a situation where unsatisfiability arises based on impossibility of some finite unfolding of the $\mathbf{U}$ formula alone (case 2) and a situation where either some finite unfolding of that formula or meeting its eventuality are possible but not both (case 4).

[3] While we defined biimplication as an abbreviation in Sect. 2, we treat it in this discussion as if it were available as an atomic operator for conjuncts of this form.

As an illustration consider the following two formulas: 1. $(\psi'\mathbf{U}\psi'') \wedge (\neg\psi' \wedge \neg\psi'')$ and 2. $(\psi'\mathbf{U}\psi'') \wedge ((\neg\psi' \wedge \neg\psi'') \vee (\mathbf{G}\neg\psi''))$ An IUC obtained without temporal unfolding will contain c, $c''' \in C'''$, and $c'''' \in C''''$ in both cases while one obtained with temporal unfolding will contain c', $c''' \in C'''$, and $c'''' \in C''''$ in the first case and additionally c'' in the second case.

Temporal unfolding leading to more fine-granular IUCs can also be applied to negative polarity occurrences of **R** formulas in a similar fashion. Application to opposite polarity occurrences for **U** and **R** as well as to negative polarity occurrences of **F** and positive polarity occurrences of **G** subformulas is possible but does not lead to more fine-granular IUCs.

Splitting Conjunctions from Temporal Unfolding. Our final variant splits the conjunctions that arise from temporal unfolding. In 4 of the 6 cases where temporal unfolding is possible this allows to distinguish the case where unsatisfiability is due to failure of unfolding in only the first time step that a **U**, **R**, **F**, or **G** formula is supposed (not) to hold in versus in the first and/or some later step. An example using an **U** formula is 1. $(\psi\mathbf{U}\psi') \wedge (\neg\psi \wedge \neg\psi')$ versus 2. $(\psi\mathbf{U}\psi') \wedge (\neg\psi' \wedge \mathbf{X}(\neg\psi \wedge \neg\psi'))$.

5.3 Comparison with Separated Normal Form

Separated Normal Form (SNF) [22, 23] is a conjunctive normal form for LTL originally proposed by Fisher to develop a resolution method for LTL.

The original SNF [22] separates past and future time operators by having a strict past time operator at the top level of the left-hand side of the implication in each conjunct and only Boolean disjunction and **F** operators on the right-hand side. We therefore restrict the comparison to two later variants [23,17] that allow propositions (present time formulas) on the left-hand side of the implications.

Compared to [22] the version of SNF in [23] also contains a simpler future time variant of SNF. [23] further refines our final variant in the last subsection in two ways. First, it applies temporal unfolding twice to **U**, weak **U**, and **G** formulas. This allows to distinguish failure of unfolding in the first, second, or some later step relative to the time when a formula is supposed to hold. Second, in some cases it has separate conjuncts for the absolute first and for later time steps. In the example $(p\mathbf{U}(q \wedge r)) \wedge ((\neg q) \wedge \mathbf{XG}\neg r)$ this allows to see that from the eventuality $q \wedge r$ the first operand is only needed in the absolute first time step, while the second operand leads to a contradiction in the second and later time steps. A minor difference is that atomic propositions are not defined using separate fresh propositions but remain unchanged at their place of occurrence.

[17] uses a less constrained version of [23]: right-hand sides of implications and bodies of **X** and **F** operators may now contain positive Boolean combinations of literals. This makes both above mentioned refinements unnecessary. The resulting normal form differs from our variant with temporal unfolding in 4 respects: 1. It works on NNF. 2. Positive Boolean combinations are not split into several conjuncts. 3. Fresh propositions are introduced for **U**, **R**, and **G** formulas representing truth in the next rather than in the current time step. Because of

that, temporal unfolding is performed at the place of occurrence of the respective $\mathbf{U}$, $\mathbf{R}$, or $\mathbf{G}$ formula. 4. As in [23] atomic propositions remain unchanged at their place of occurrence. The combination of 2 and 4 leads to this variant of SNF yielding less information in the following example: $(\mathbf{F}(p \wedge q)) \wedge \mathbf{G}\neg p$. An IUC resulting from this variant of SNF will contain the conjunct $x \rightarrow \mathbf{F}(p \wedge q)$, not making it clear that q is irrelevant for unsatisfiability. On the other hand, unsatisfiability due to failure of temporal unfolding at the first time point only can in some cases be distinguished from that at the first and/or or later time points, thus yielding more information; $(\mathbf{G}p) \wedge \neg p$ is an example for that.

6 Unsatisfiable Cores via Bounded Model Checking

By encoding existence of counterexamples of bounded length into a set of CNF clauses SAT-based Bounded Model Checking (BMC) (e.g., [6]) reduces model checking of LTL to SAT. Details on BMC can be found, e.g., in [7].

To prove correctness of properties (rather than existence of a counterexample) BMC needs to determine when to stop searching for longer counterexamples. The original works (e.g., [6]) imposed an upper bound derived from the graph structure of the model. A more refined method (e.g., [42]) takes a two-step approach: For the current bound on the length of counterexamples k, check whether there exists a path that 1. could possibly be extended to form a counterexample to the property and 2. contains no redundant part. If either of the two checks fails and no counterexample of length $\leq k$ has been found, then declare correctness of the property. As there are only finitely many states, step 2 guarantees termination. For other methods to prove properties in BMC see, e.g., [7].

By assuming a universal model BMC provides a way to determine LTL satisfiability (used, e.g., in [16]) and so is a natural choice to investigate notions of UCs. Note that in BMC, as soon as properties are not just simple invariants of the form $\mathbf{G}p$, already the first part of the above check for termination might fail. That observation yields an incomplete method to determine LTL satisfiability. We first sketch the method and then the UCs that can be extracted.

The method essentially employs dCNF with splitting conjunctions from temporal unfolding to generate a SAT problem in CNF as follows: 1. Pick some bound k. 2. To obtain the set of variables instantiate the members of X for each time step $0 \leq i \leq k+1$ and of AP for $0 \leq i \leq k$. We indicate the time step by using superscripts. 3. For the set of CNF clauses instantiate each conjunct in $dCNF_{aux}$ not containing a $\mathbf{F}$ or $\mathbf{G}$ operator once for each $0 \leq i \leq k$. Add the time 0 instance of the root of the dCNF, x_{ϕ}^{0}, to the set of clauses. 4. Replace each occurrence of $\mathbf{X}x_{\psi}^{i}$ with x_{ψ}^{i+1}. Note that at this point all temporal operators have been removed and we indeed have a CNF. Now if for any such k the resulting CNF is unsatisfiable, then so is the original LTL formula. The resulting method is very similar to BMC in [29] when checking for termination by using the completeness formula only rather than completeness and simplepath formula together (only presence of the latter can ensure termination).

	x^0_ϕ		
✔	$\boxed{(x^0_\phi \rightarrow x^0_{p\vee\mathbf{XX}p})}$	$(x^1_\phi \rightarrow x^1_{p\vee\mathbf{XX}p})$	$(x^2_\phi \rightarrow x^2_{p\vee\mathbf{XX}p})$
✔	$\boxed{(x^0_{p\vee\mathbf{XX}p} \rightarrow x^0_{p,0} \vee x^0_{\mathbf{XX}p})}$	$(x^1_{p\vee\mathbf{XX}p} \rightarrow x^1_{p,0} \vee x^1_{\mathbf{XX}p})$	$(x^2_{p\vee\mathbf{XX}p} \rightarrow x^2_{p,0} \vee x^2_{\mathbf{XX}p})$
✔	$\boxed{(x^0_{p,0} \rightarrow p)}$	$(x^1_{p,0} \rightarrow p)$	$(x^2_{p,0} \rightarrow p)$
✔	$\boxed{(x^0_{\mathbf{XX}p} \rightarrow x^1_{\mathbf{X}p})}$	$(x^1_{\mathbf{XX}p} \rightarrow x^2_{\mathbf{X}p})$	$(x^2_{\mathbf{XX}p} \rightarrow x^3_{\mathbf{X}p})$
✔	$(x^0_{\mathbf{X}p} \rightarrow x^1_{p,1})$	$\boxed{(x^1_{\mathbf{X}p} \rightarrow x^2_{p,1})}$	$(x^2_{\mathbf{X}p} \rightarrow x^3_{p,1})$
✔	$(x^0_{p,1} \rightarrow p)$	$(x^1_{p,1} \rightarrow p)$	$\boxed{(x^2_{p,1} \rightarrow p)}$
✔	$\boxed{(x^0_\phi \rightarrow x^0_{\mathbf{G}(\neg p\wedge q)})}$	$(x^1_\phi \rightarrow x^1_{\mathbf{G}(\neg p\wedge q)})$	$(x^2_\phi \rightarrow x^2_{\mathbf{G}(\neg p\wedge q)})$
✔	$\boxed{(x^0_{\mathbf{G}(\neg p\wedge q)} \rightarrow x^1_{\mathbf{G}(\neg p\wedge q)})}$	$\boxed{(x^1_{\mathbf{G}(\neg p\wedge q)} \rightarrow x^2_{\mathbf{G}(\neg p\wedge q)})}$	$(x^2_{\mathbf{G}(\neg p\wedge q)} \rightarrow x^3_{\mathbf{G}(\neg p\wedge q)})$
✔	$\boxed{(x^0_{\mathbf{G}(\neg p\wedge q)} \rightarrow x^0_{\neg p\wedge q})}$	$(x^1_{\mathbf{G}(\neg p\wedge q)} \rightarrow x^1_{\neg p\wedge q})$	$\boxed{(x^2_{\mathbf{G}(\neg p\wedge q)} \rightarrow x^2_{\neg p\wedge q})}$
✔	$\boxed{(x^0_{\neg p\wedge q} \rightarrow x^0_{\neg p})}$	$(x^1_{\neg p\wedge q} \rightarrow x^1_{\neg p})$	$\boxed{(x^2_{\neg p\wedge q} \rightarrow x^2_{\neg p})}$
✔	$\boxed{(x^0_{\neg p} \rightarrow \neg x^0_{p,2})}$	$(x^1_{\neg p} \rightarrow \neg x^1_{p,2})$	$\boxed{(x^2_{\neg p} \rightarrow \neg x^2_{p,2})}$
✔	$\boxed{(\neg x^0_{p,2} \rightarrow \neg p)}$	$(\neg x^1_{p,2} \rightarrow \neg p)$	$\boxed{(\neg x^2_{p,2} \rightarrow \neg p)}$
	$(x^0_{\neg p\wedge q} \rightarrow x^0_q)$	$(x^1_{\neg p\wedge q} \rightarrow x^1_q)$	$(x^2_{\neg p\wedge q} \rightarrow x^2_q)$
	$(x^0_q \rightarrow q)$	$(x^1_q \rightarrow q)$	$(x^2_q \rightarrow q)$
dCNF core	time step 0	time step 1	time step 2

Fig. 3. Example of an UC via BMC. The input formula is $\phi = (p \vee \mathbf{XX}p) \wedge \mathbf{G}(\neg p \wedge q)$. Clauses that form the SAT IUC are marked blue boxed. A tick in the leftmost column indicates that the corresponding dCNF clause is part of a UC via dCNF.

Assume that for an LTL formula ϕ the above method yields an unsatisfiable CNF for some k and that we are provided with an IUC of that CNF as a subset of clauses. It is easy to see that we can extract an UC of the granularity of a dCNF with splitting conjunctions from temporal unfolding by considering any dCNF conjunct to be part of the UC iff for any time step the corresponding CNF clause is present in the CNF IUC. Note that the CNF IUC provides potentially finer granularity in the temporal dimension: the CNF IUC contains information about the relevance of parts of the LTL formula to unsatisfiability at each time step. Contrary to the notions of UC in the previous section (see [41]) we currently have no translation back to LTL for this level of detail. Once such translation has been obtained it makes sense to define removal of clauses from the CNF as the operation to derive a core thus giving the notions of core, unsatisfiable core, proper unsatisfiable core, and irreducible unsatisfiable core via BMC.

As an example consider $\phi = (p \vee \mathbf{XX}p) \wedge \mathbf{G}(\neg p \wedge q)$. The translation into a set of CNF clauses and the CNF IUC are depicted in Fig. 3. Extracting an UC at the granularity of a dCNF with splitting conjunctions from temporal unfolding results in a dCNF equivalent to $(p \vee \mathbf{XX}p) \wedge \mathbf{G}(\neg p \wedge 1)$. The CNF IUC shows that the occurrence of $\neg p$ is relevant only at time steps 0 and 2.

7 Unsatisfiable Cores via Tableaux

Tableaux are widely used for temporal logics. Most common methods in BDD-based symbolic model checking (e.g., [19]) and in explicit state model checking (e.g., [25]) of LTL rely on tableaux. Therefore tableaux seem to be a natural candidate for investigating notions of UCs.

In this section we only consider formulas in NNF. We assume that the reader is familiar with standard tableaux constructions for LTL such as [25]. We differ from, e.g., [25] in that we retain and continue to expand closed nodes during tableau construction and only take them into account when searching for satisfied paths in the tableau. We fix some terminology. A node in a tableau is called 1. *initial* if it is a potential start, 2. *closed* if it contains a pair of contradicting literals or 0, 3. *terminal* if it contains no obligations left for the next time step, and 4. *accepting* (for some **U** or **F** formula), if it either contains both the formula and its eventuality or none of the two. A path in the tableau is *initialized* if it starts in an initial node and *fair* if it contains infinitely many occurrences of accepting nodes for each **U** and **F** formula. A path is *satisfied* if 1. it is initialized, 2. it contains no closed node, and 3. it is finite and ends in a terminal node or infinite and fair. A tableau is *satisfied* iff it contains a satisfied path.

Intuitively, closed nodes are what prevents satisfied paths. For an initialized path to a terminal node it is obvious that a closed node on that path is a reason for that path not being satisfied. A similar statement holds for initialized infinite fair paths that contain closed nodes. That leaves initialized infinite unfair paths that do not contain a closed node. Still, also in that case closed nodes hold information w.r.t. non-satisfaction: an unfair path contains at least one occurrence of an **U** or **F** formula whose eventuality is not fulfilled. The tableau construction ensures that for each node containing such an occurrence there will also be a branch that attempts to make the eventuality 1. That implies that the reason for failure of fulfilling eventualities is not to be found on the infinite unfair path, but on its unsuccessful branches. Hence, we focus on closed nodes to extract sufficient information why a formula in unsatisfiable.

The procedure to extract an UC now works as follows. It first chooses a subset of closed nodes that act as a barrier in that at least one of these nodes is in the way of each potentially satisfied path in the tableau. Next it chooses a set of occurrences of contradicting literals and 0 s.t. this set represents a contradiction for each of the selected closed tableau nodes. As these occurrences of subformulas make up the reason for non-satisfaction, they and, transitively, their fathers in the parse tree of the formula are marked and retained while all non-marked occurrences of subformulas in the parse tree are discarded and dangling edges are rerouted to fresh nodes representing 1.

As an example consider the tableau in Fig. 4 for $\phi = \mathbf{X}(((\mathbf{G}(p \wedge q \wedge r)) \wedge (\mathbf{F}(\neg p \wedge \neg q))) \vee (p \wedge (\mathbf{X}p) \wedge \neg p \wedge \mathbf{X}(\neg p)))$. Choosing $\{n_1, n_3\}$ as the subset of closed nodes and the occurrences of q, $\neg q$ in n_1 and p, $\neg p$ in n_3 leads to $\mathbf{X}(((\mathbf{G}(1 \wedge q \wedge 1)) \wedge (\mathbf{F}(1 \wedge \neg q))) \vee (p \wedge 1 \wedge \neg p \wedge 1))$ as UC. More UCs result from choosing p, $\neg p$ also in n_1, or n_5 instead of n_3.

In the full version [41] we show that the set of UCs that can be extracted in that way is equivalent to the set of UCs via parse trees. However, we conjecture that the procedure can be extended to extract UCs that indicate relevance of subformulas not only at finitely many time steps as in Sect. 6 but at semilinearly many. Given, e.g., $\phi = p \wedge (\mathbf{G}(p \rightarrow \mathbf{XX}p)) \wedge (\mathbf{F}(\neg p \wedge \mathbf{X}\neg p))$, we would like to see that some subformulas are only relevant at every second time step.

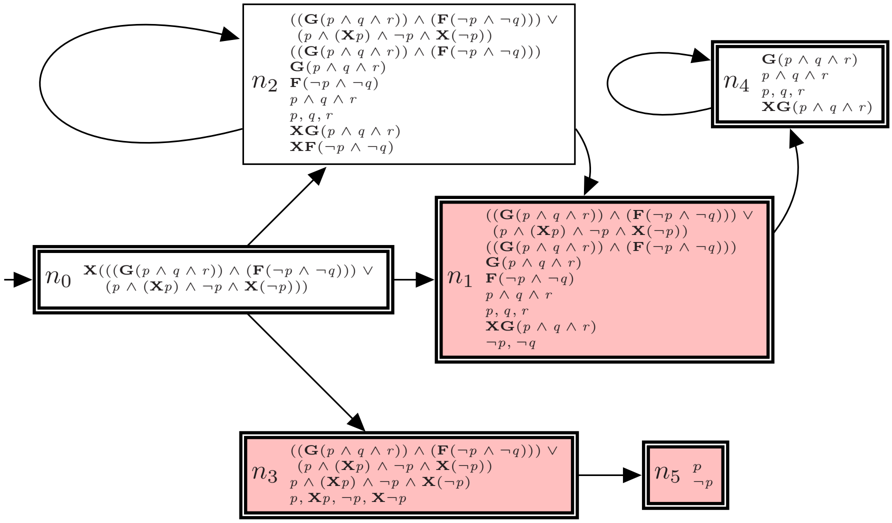

n_2
$((\mathbf{G}(p \wedge q \wedge r)) \wedge (\mathbf{F}(\neg p \wedge \neg q))) \vee$
$(p \wedge (\mathbf{X}p) \wedge \neg p \wedge \mathbf{X}(\neg p))$
$((\mathbf{G}(p \wedge q \wedge r)) \wedge (\mathbf{F}(\neg p \wedge \neg q)))$
$\mathbf{G}(p \wedge q \wedge r)$
$\mathbf{F}(\neg p \wedge \neg q)$
$p \wedge q \wedge r$
p, q, r
$\mathbf{XG}(p \wedge q \wedge r)$
$\mathbf{XF}(\neg p \wedge \neg q)$
n_4
$\mathbf{G}(p \wedge q \wedge r)$
$p \wedge q \wedge r$
p, q, r
$\mathbf{XG}(p \wedge q \wedge r)$
n_0
$\mathbf{X}(((\mathbf{G}(p \wedge q \wedge r)) \wedge (\mathbf{F}(\neg p \wedge \neg q))) \vee$
$(p \wedge (\mathbf{X}p) \wedge \neg p \wedge \mathbf{X}(\neg p)))$
n_1
$((\mathbf{G}(p \wedge q \wedge r)) \wedge (\mathbf{F}(\neg p \wedge \neg q))) \vee$
$(p \wedge (\mathbf{X}p) \wedge \neg p \wedge \mathbf{X}(\neg p))$
$((\mathbf{G}(p \wedge q \wedge r)) \wedge (\mathbf{F}(\neg p \wedge \neg q)))$
$\mathbf{G}(p \wedge q \wedge r)$
$\mathbf{F}(\neg p \wedge \neg q)$
$p \wedge q \wedge r$
p, q, r
$\mathbf{XG}(p \wedge q \wedge r)$
$\neg p, \neg q$
n_3
$((\mathbf{G}(p \wedge q \wedge r)) \wedge (\mathbf{F}(\neg p \wedge \neg q))) \vee$
$(p \wedge (\mathbf{X}p) \wedge \neg p \wedge \mathbf{X}(\neg p))$
$p \wedge (\mathbf{X}p) \wedge \neg p \wedge \mathbf{X}(\neg p)$
$p, \mathbf{X}p, \neg p, \mathbf{X}\neg p$
n_5
p
$\neg p$

Fig. 4. Example of an unsatisfied tableau for $\phi = \mathbf{X}(((\mathbf{G}(p \wedge q \wedge r)) \wedge (\mathbf{F}(\neg p \wedge \neg q))) \vee (p \wedge (\mathbf{X}p) \wedge \neg p \wedge \mathbf{X}(\neg p)))$. The initial node n_0 has an incoming arrow, closed nodes n_1, n_3, n_5 are filled red, accepting nodes (all but n_2) have thick double lines, and the terminal node n_5 has no outgoing arrow.

8 Related Work

Notions of Core. [16] proposes a notion of UCs of LTL formulas. The context in that work is a method for satisfiability checking of LTL formulas by using Boolean abstraction (e.g., [30]). As a consequence, an UC in [16] is a subset of the set of top-level temporal formulas, potentially leading to very coarse cores.

SAT uses CNF as a standard format and UCs are typically subsets of clauses (e.g., [9]). Similarly, in constraint programming, an UC is a subset of the set of input constraints (e.g., [4]); [27] suggests a more fine-grained notion based on unsatisfiable tuples. Finally, also in SMT UCs are subsets of formulas (e.g., [14]).

For realizability [38] of a set of LTL formulas, partitioned into a set of assumptions and a set of guarantees, [15] suggests to first reduce the number of guarantees and then, additionally, to reduce the set of assumptions.

Extracting Cores from Proofs. In [34] a successful run of a model checker, which essentially corresponds to an unsatisfied tableau, is used to extract a temporal proof from the tableau [25] as a certificate that the model fulfills the specification. [32] generates certificates for successful model checking runs of μ-calculus specifications. [40] extracts UCs from unsatisfied tableaux to aid debugging in the context of description logics. Extracting a core from a resolution proof is an established technique in propositional SAT (e.g., [26,48]). In SMT UCs from

SAT can be used to extract UCs for SMT [14]. Extraction from proofs is also used in vacuity checking [33, 44].

Applications of Cores. Using UCs to help a user debugging by pointing out a subset of the input as part of some problem is stated explicitly as motivation in many works on cores, e.g., [10, 4, 9, 48].

[43] presents a method for debugging declarative specifications by translating an abstract syntax tree (AST) of an inconsistent specification to CNF, extracting an UC from the CNF, and mapping the result back to AST highlighting only the relevant parts. That work has some similarities with our discussion; however, there are also a number of differences. 1. The exposition in [43] is for first order relational logic and generalizes to languages that are reducible to SAT, while our logic is LTL. 2. The motivation and focus of [43] is on the method of core extraction, and it is accompanied by some experimental results. The notion of a core as parts of the AST is taken as a given. On the other hand, our focus is on investigating different notions of cores and on comparing the resulting information that can be gained. 3. [43] does not consider tableaux. [45] suggests improved algorithms for core extraction compared to [43]; the improved algorithms produce IUCs at a reasonable cost by using mechanisms similar to [48, 21]. The scope of the method is extended to specification languages with a (restricted) translation to logics with resolution engine.

Examples of using UCs for debugging in description logics and ontologies are [40, 46]. For temporal logic, the methodology proposed in [35] suggests to return a subset of the specification in case of a problem. For [15] see above.

The application of UCs as filters in an iterative search is mentioned in Sect. 1.

Vacuity Checking. Vacuity checking (e.g., [5,31]) is a technique in model checking to determine whether a model satisfies the specification in an undesired way. Vacuity asks whether there exists a strengthening of a specification s.t. the model still passes that strengthened specification. The original notion of vacuity from [5, 31] replaces occurrences of subformulas in the specification with 0 or 1 depending on polarity and is, therefore, related to the notion of UC in Sect. 4.

The comparison of notions of vacuity with UCs is as follows: 1. Vacuity is normally defined with respect to a specific model. [13] proposes vacuity without design as a preliminary check of vacuity: a formula is vacuous without design if it fulfills a variant of itself to which a strengthening operation has been applied. [24] extends that into a framework for inherent vacuity (see below). 2. Vacuity is geared to answer whether there exists at least one strengthening of the specification s.t. the model still satisfies the specification. For that it is sufficient to demonstrate that with a single strengthening step. The question of whether and to which extent the specification should be strengthened is then usually left to the designer. In core extraction one would ideally like to obtain IUCs and do so in a fully automated fashion. [28, 13] discuss mutual vacuity, i.e., vacuity w.r.t. sets of subformulas. [11] proceeds to obtain even stronger passing formulas combining several strengthened versions of the original formula. 3. Vacuity typically focuses on strengthening a formula while methods to obtain UCs use weakening. The reason is that in the case of a failing specification a counterexample is considered to be more helpful. Still, vacuity is defined in, e.g., [5, 31, 24] w.r.t. both passing and failing formulas.

[24] proposes a framework to identify inherent vacuity, i.e., specifications that are vacuous in any model. The framework has 4 parameters: 1. vacuity type: occurrences of subformulas, sharing of subformulas, etc., 2. equivalence type: closed or open systems, 3. tightening type: equivalence or preservance of satisfiability/realizability, and 4. polarity type: strengthening or weakening. Our notion of UCs via parse tree is very closely related to the following instance of that framework. Let the vacuity type be that of replacing occurrences of subformulas with 1 or 0 depending on polarity [5], systems be closed, tightening type be equivalence or preservance of unsatisfiability, and polarity type be weakening. Then it is straightforward to show that, given a proper UC ϕ' via parse tree of some unsatisfiable formula ϕ, 1. ϕ is inherently vacuous, and 2. ϕ' is an IUC iff it is not inherently vacuous. [24] focuses on satisfiable/realizable instances and doesn't make a connection to the notion of unsatisfiable or unrealizable cores.

[44] exploits resolution proofs from BMC runs in order to extract information on vacuity including information on relevance of subformulas at specific time steps in a fashion related to our extraction of UCs in Sect. 6. A difference is that the presentation in [44] only explains how to obtain the notion of k-step vacuity from some BMC run with bound k but leaves it unclear how to make the transition from the notion of k-step vacuity to the notion of vacuity and, similarly, how to aggregate results on the relevance of subformulas at specific time steps over results for different ks; our method of UC extraction can return an UC as soon as the generated CNF is unsatisfiable for some k.

[39] suggests to generalize the operations to strengthen a specification by considering a form of interpolants between a model and its specification. While this might lead to another possibility to derive a core from a formula, an arbitrary interpolant might not allow the user to easily see what is happening. Hence, [39] needs to be concretized to meet that criterion.

Other notions and techniques might be suitable to be carried over from vacuity detection to UCs for LTL and vice versa. E.g., [2] extends vacuity to consider sharing of subformulas. We are not aware of any work in vacuity that takes the perspective of searching an UC of an LTL formula or considers dCNFs as we do.

9 Conclusion

We suggested notions of unsatisfiable cores for LTL formulas that provide strictly more fine-grained information than the (few) previous notions. While basic notions turned out to be equivalent, some variants were shown to provide or potentially provide more information, in particular, in the temporal dimension.

We stated initially that we see methods of UC extraction as a means to suggest notions of UCs. Indeed, it turned out that each method for core extraction suggested a different or a more fine-grained notion of UC that should be taken into account. It seems to be likely, though, that some of the more fine-grained notions can be obtained also with other UC extraction methods.

Directions for future work include defining and obtaining the more fine-grained notions of UC suggested at the end of Sect.s 6 and 7, investigating the notion of UC that results from temporal resolution proofs, taking sharing of subformulas into account, and extending the notions to realizability. Equally important are efficient implementations. Finally, while in theory two algorithms to obtain UCs might be able to come up with the same set of UCs, practical implementations could yield different UCs due to how non-determinism is resolved; hence, an empirical evaluation of the usefulness of the resulting UCs is needed.

Acknowledgements. The author thanks the research groups at FBK and Verimag for discussions and comments, esp., A. Cimatti, M. Roveri, and S. Tonetta. Part of this work was carried out while the author was at Verimag/CNRS. He thanks O. Maler for providing the freedom to pursue this work. Finally, the author thanks the Provincia Autonoma di Trento for support (project EMTELOS).

References

1. Prosyd, `http://www.prosyd.org/`
2. Armoni, R., Fix, L., Flaisher, A., Grumberg, O., Piterman, N., Tiemeyer, A., Vardi, M.: Enhanced vacuity detection in linear temporal logic. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 368–380. Springer, Heidelberg (2003)
3. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
4. Bakker, R., Dikker, F., Tempelman, F.: Diagnosing and solving over-determined constraint satisfaction problems. In: IJCAI (1993)
5. Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient detection of vacuity in temporal model checking. Formal Methods in System Design 18(2) (2001)
6. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, p. 193. Springer, Heidelberg (1999)
7. Biere, A., Heljanko, K., Junttila, T., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. Logical Methods in Computer Science 2(5) (2006)
8. Bloem, R., Cavada, R., Pill, I., Roveri, M., Tchaltsev, A.: RAT: A tool for the formal analysis of requirements. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 263–267. Springer, Heidelberg (2007)
9. Bruni, R., Sassano, A.: Restoring satisfiability or maintaining unsatisfiability by finding small unsatisfiable subformulae. In: SAT (2001)
10. Chinneck, J., Dravnieks, E.: Locating minimal infeasible constraint sets in linear programs. ORSA Journal on Computing 3(2) (1991)
11. Chockler, H., Gurfinkel, A., Strichman, O.: Beyond vacuity: Towards the strongest passing formula. In: FMCAD (2008)
12. Chockler, H., Kupferman, O., Vardi, M.: Coverage metrics for temporal logic model checking. Formal Methods in System Design 28(3) (2006)
13. Chockler, H., Strichman, O.: Easier and more informative vacuity checks. In: MEMOCODE (2007)

14. Cimatti, A., Griggio, A., Sebastiani, R.: A simple and flexible way of computing small unsatisfiable cores in SAT modulo theories. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 334–339. Springer, Heidelberg (2007)
15. Cimatti, A., Roveri, M., Schuppan, V., Tchaltsev, A.: Diagnostic information for realizability. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 52–67. Springer, Heidelberg (2008)
16. Cimatti, A., Roveri, M., Schuppan, V., Tonetta, S.: Boolean abstraction for temporal logic satisfiability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 532–546. Springer, Heidelberg (2007)
17. Cimatti, A., Roveri, M., Sheridan, D.: Bounded verification of past LTL. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 245–259. Springer, Heidelberg (2004)
18. Cimatti, A., Roveri, M., Susi, A., Tonetta, S.: From informal requirements to property-driven formal validation. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 166–181. Springer, Heidelberg (2009)
19. Clarke, E., Grumberg, O., Hamaguchi, K.: Another look at LTL model checking. Formal Methods in System Design 10(1) (1997)
20. Clarke, E., Talupur, M., Veith, H., Wang, D.: SAT based predicate abstraction for hardware verification. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 78–92. Springer, Heidelberg (2004)
21. Dershowitz, N., Hanna, Z., Nadel, A.: A scalable algorithm for minimal unsatisfiable core extraction. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 36–41. Springer, Heidelberg (2006)
22. Fisher, M.: A resolution method for temporal logic. In: IJCAI (1991)
23. Fisher, M., Dixon, C., Peim, M.: Clausal temporal resolution. ACM Trans. Comput. Log. 2(1) (2001)
24. Fisman, D., Kupferman, O., Sheinvald-Faragy, S., Vardi, M.: A framework for inherent vacuity. In: Chockler, H., Hu, A.J. (eds.) HVC 2008. LNCS, vol. 5394, pp. 7–22. Springer, Heidelberg (2009)
25. Gerth, R., Peled, D., Vardi, M., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: PSTV (1995)
26. Goldberg, E., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: DATE (2003)
27. Grégoire, É., Mazure, B., Piette, C.: MUST: Provide a finer-grained explanation of unsatisfiability. In: Bessiere, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 317–331. Springer, Heidelberg (2007)
28. Gurfinkel, A., Chechik, M.: How vacuous is vacuous? In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 451–466. Springer, Heidelberg (2004)
29. Heljanko, K., Junttila, T., Latvala, T.: Incremental and complete bounded model checking for full PLTL. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 98–111. Springer, Heidelberg (2005)
30. Kroening, D., Strichman, O.: Decision Procedures. Springer, Heidelberg (2008)
31. Kupferman, O., Vardi, M.: Vacuity detection in temporal model checking. STTT 4(2) (2003)
32. Namjoshi, K.: Certifying model checkers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, p. 2. Springer, Heidelberg (2001)
33. Namjoshi, K.: An efficiently checkable, proof-based formulation of vacuity in model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 57–69. Springer, Heidelberg (2004)

34. Peled, D., Pnueli, A., Zuck, L.: From falsification to verification. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS 2001. LNCS, vol. 2245, p. 292. Springer, Heidelberg (2001)
35. Pill, I., Semprini, S., Cavada, R., Roveri, M., Bloem, R., Cimatti, A.: Formal analysis of hardware requirements. In: DAC (2006)
36. Plaisted, D., Greenbaum, S.: A structure-preserving clause form translation. J. Symb. Comput. 2(3) (1986)
37. Pnueli, A.: The temporal logic of programs. In: FOCS (1977)
38. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. POPL (1989)
39. Samer, M., Veith, H.: On the notion of vacuous truth. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 2–14. Springer, Heidelberg (2007)
40. Schlobach, S., Cornet, R.: Non-standard reasoning services for the debugging of description logic terminologies. In: IJCAI. Morgan Kaufmann, San Francisco (2003)
41. Schuppan, V.: Towards a notion of unsatisfiable cores for LTL. Technical Report 200901000, Fondazione Bruno Kessler (2009)
42. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)
43. Shlyakhter, I., Seater, R., Jackson, D., Sridharan, M., Taghdiri, M.: Debugging overconstrained declarative models using unsatisfiable cores. In: ASE (2003)
44. Simmonds, J., Davies, J., Gurfinkel, A., Chechik, M.: Exploiting resolution proofs to speed up LTL vacuity detection for BMC. In: FMCAD (2007)
45. Torlak, E., Chang, F., Jackson, D.: Finding minimal unsatisfiable cores of declarative specifications. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 326–341. Springer, Heidelberg (2008)
46. Wang, H., Horridge, M., Rector, A., Drummond, N., Seidenberg, J.: Debugging OWL-DL ontologies: A heuristic approach. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) ISWC 2005. LNCS, vol. 3729, pp. 745–757. Springer, Heidelberg (2005)
47. Wolfman, S., Weld, D.: The LPSAT engine & its application to resource planning. In: IJCAI. Morgan Kaufmann, San Francisco (1999)
48. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable Boolean formula. Presented at SAT (2003)

Rule Formats for Determinism and Idempotence*

Luca Aceto[1], Arnar Birgisson[1], Anna Ingolfsdottir[1],
MohammadReza Mousavi[2], and Michel A. Reniers[2]

[1] School of Computer Science, Reykjavik University,
Kringlan 1, IS-103 Reykjavik, Iceland
[2] Department of Computer Science, Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

Abstract. Determinism is a semantic property of (a fragment of) a language that specifies that a program cannot evolve operationally in several different ways. Idempotence is a property of binary composition operators requiring that the composition of two identical specifications or programs will result in a piece of specification or program that is equivalent to the original components. In this paper, we propose two (related) meta-theorems for guaranteeing determinism and idempotence of binary operators. These meta-theorems are formulated in terms of syntactic templates for operational semantics, called rule formats. We show the applicability of our formats by applying them to various operational semantics from the literature.

1 Introduction

Structural Operational Semantics (SOS) [18] is a popular method for assigning a rigorous meaning to specification and programming languages. The meta-theory of SOS provides powerful tools for proving semantic properties for such languages without investing too much time on the actual proofs; it offers syntactic templates for SOS rules, called *rule formats*, which guarantee semantic properties once the SOS rules conform to the templates (see, e.g., the references [1,16] for surveys on the meta-theory of SOS). There are various rule formats in the literature for many different semantic properties, ranging from basic properties such as commutativity [15] and associativity [6] of operators, and congruence of behavioral equivalences (see, e.g., [22]) to more technical and involved ones such as non-interference [19] and (semi-)stochasticity [12]. In this paper, we propose rule formats for two (related) properties, namely, determinism and idempotence.

Determinism is a semantic property of (a fragment of) a language that specifies that a program cannot evolve operationally in several different ways. It holds

* The work of Aceto, Birgisson and Ingolfsdottir has been partially supported by the projects "The Equational Logic of Parallel Processes" (nr. 060013021), and "New Developments in Operational Semantics" (nr. 080039021) of the Icelandic Research Fund. Birgisson has been further supported by a research-student grant nr. 080890008 of the Icelandic Research Fund.

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 146–161, 2010.

for sub-languages of many process calculi and programming languages, and it is also a crucial property for many formalisms for the description of timed systems, where time transitions are required to be deterministic, because the passage of time should not resolve any choice.

Idempotence is a property of binary composition operators requiring that the composition of two identical specifications or programs will result in a piece of specification or program that is equivalent to the original components. Idempotence of a binary operator f is concisely expressed by the following algebraic equation.

$$f(x, x) = x$$

Determinism and idempotence may seem unrelated at first sight. However, it turns out that in order to obtain a powerful rule format for idempotence, we need to have the determinism of certain transition relations in place. Therefore, having a syntactic condition for determinism, apart from its intrinsic value, results in a powerful, yet syntactic framework for idempotence.

To our knowledge, our rule format for idempotence has no precursor in the literature. As for determinism, in [8], a rule format for bounded nondeterminism is presented but the case for determinism is not studied. Also, in [20] a rule format is proposed to guarantee several time-related properties, including time determinism, in the settings of Ordered SOS. In case of time determinism, their format corresponds to a subset of our rule format when translated to the setting of ordinary SOS, by means of the recipe given in [13].

We made a survey of existing deterministic process calculi and of idempotent binary operators in the literature and we have applied our formats to them. Our formats could cover all practical cases that we have discovered so far, which is an indication of its expressiveness and relevance.

The rest of this paper is organized as follows. In Section 2 we recall some basic definitions from the meta-theory of SOS. In Section 3, we present our rule format for determinism and prove that it does guarantee determinism for certain transition relations. Section 4 introduces a rule format for idempotence and proves it correct. In Sections 3 and 4, we also provide several examples to motivate the constraints of our rule formats and to demonstrate their practical applications. Finally, Section 5 concludes the paper and presents some directions for future research.

2 Preliminaries

In this section we present, for sake of completeness, some standard definitions from the meta-theory of SOS that will be used in the remainder of the paper.

Definition 1 (Signature and Terms). *We let V represent an infinite set of variables and use $x, x', x_i, y, y', y_i, \ldots$ to range over elements of V. A* signature *Σ is a set of function symbols, each with a fixed arity. We call these symbols* operators *and usually represent them by $f, g, \ldots$. An operator with arity zero is called a* constant*. We define the set $\mathbb{T}(\Sigma)$ of* terms *over Σ as the smallest set satisfying the following constraints.*

- *A variable $x \in V$ is a term.*
- *If $f \in \Sigma$ has arity n and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.*

We use $t, t', t_i, \ldots$ to range over terms. We write $t_1 \equiv t_2$ if t_1 and t_2 are syntactically equal. The function vars : $\mathbb{T}(\Sigma) \to 2^V$ gives the set of variables appearing in a term. The set $\mathbb{C}(\Sigma) \subseteq \mathbb{T}(\Sigma)$ is the set of closed terms*, i.e., terms that contain no variables. We use $p, p', p_i, \ldots$ to range over closed terms. A* substitution *σ is a function of type $V \to \mathbb{T}(\Sigma)$. We extend the domain of substitutions to terms homomorphically. If the range of a substitution lies in $\mathbb{C}(\Sigma)$, we say that it is a* closing substitution*.*

Definition 2 (Transition System Specifications (TSS), Formulae and Transition Relations). *A* transition system specification *is a triplet (Σ, L, D) where*

- *Σ is a signature.*
- *L is a set of labels. If $l \in L$, and $t, t' \in \mathbb{T}(\Sigma)$ we say that $t \xrightarrow{l} t'$ is a* positive formula *and $t \overset{l}{\nrightarrow}$ is a* negative formula*. A formula, typically denoted by ϕ, ψ, ϕ', ϕ_i, ... is either a negative formula or a positive one.*
- *D is a set of* deduction rules*, i.e., tuples of the form (Φ, ϕ) where Φ is a set of formulae and ϕ is a positive formula. We call the formulae contained in Φ the* premises *of the rule and ϕ the* conclusion*.*

We write vars(r) to denote the set of variables appearing in a deduction rule (r)*. We say that a formula is* closed *if all of its terms are closed. Substitutions are also extended to formulae and sets of formulae in the natural way. A set of positive closed formulae is called a* transition relation*.*

We often refer to a formula $t \xrightarrow{l} t'$ as a *transition* with t being its *source*, l its label, and t' its *target*. A deduction rule (Φ, ϕ) is typically written as $\frac{\Phi}{\phi}$. For a deduction rule r, we write *conc*(r) to denote its conclusion and *prem*(r) to denote its premises. We call a deduction rule *f-defining* when the outermost function symbol appearing in the source of its conclusion is f.

The meaning of a TSS is defined by the following notion of least three-valued stable model. To define this notion, we need two auxiliary definitions, namely provable transition rules and contradiction, which are given below.

Definition 3 (Provable Transition Rules). *A deduction rule is called a* transition rule *when it is of the form $\frac{N}{\phi}$ with N a set of* negative formulae*. A TSS $\mathcal{T}$* proves $\frac{N}{\phi}$*, denoted by $\mathcal{T} \vdash \frac{N}{\phi}$, when there is a well-founded upwardly branching tree with formulae as nodes and of which*

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

Definition 4 (Contradiction and Contingency). *Formula $t \xrightarrow{l} t'$ is said to* contradict $t \overset{l}{\nrightarrow}$*, and vice versa. For two sets Φ and Ψ of formulae, Φ* contradicts

Ψ, denoted by $\Phi \nvDash \Psi$, when there is a $\phi \in \Phi$ that contradicts a $\psi \in \Psi$. Φ is contingent *w.r.t. Ψ, denoted by $\Phi \vDash \Psi$, when Φ does not contradict Ψ.*

It immediately follows from the above definition that contradiction and contingency are symmetric relations on (sets of) formulae. We now have all the necessary ingredients to define the semantics of TSSs in terms of three-valued stable models.

Definition 5 (The Least Three-Valued Stable Model). *A pair (C, U) of sets of positive closed transition formulae is called a* three-valued stable model *for a TSS $\mathcal{T}$ when*

- *for all $\phi \in C$, $\mathcal{T} \vdash \frac{N}{\phi}$ for a set N such that $C \cup U \vDash N$, and*
- *for all $\phi \in U$, $\mathcal{T} \vdash \frac{N}{\phi}$ for a set N such that $C \vDash N$.*

C stands for Certainly *and U for* Unknown*; the third value is determined by the formulae not in $C \cup U$. The* least *three-valued stable model is a three-valued stable model which is the least with respect to the ordering on pairs of sets of formulae defined as $(C, U) \leq (C', U')$ iff $C \subseteq C'$ and $U' \subseteq U$. When for the least three-valued stable model it holds that $U = \emptyset$, we say that $\mathcal{T}$ is* complete.

Complete TSSs univocally define a transition relation, i.e., the C component of their least three-valued stable model. Completeness is central to almost all meta-results in the SOS meta-theory and, as it turns out, it also plays an essential role in our meta-results concerning determinism and idempotence. All practical instances of TSSs are complete and there are syntactic sufficient conditions guaranteeing completeness, see for example [9].

3 Determinism

Definition 6 (Determinism). *A transition relation T is called deterministic for label l, when if $p \xrightarrow{l} p' \in T$ and $p \xrightarrow{l} p'' \in T$, then $p' \equiv p''$.*

Before we define a format for determinism, we need two auxiliary definitions. The first one is the definition of source dependent variables, which we borrow from [14] with minor additions.

Definition 7 (Source Dependency). *For a deduction rule, we define the set of* source dependent *variables as the smallest set that contains*

1. *all variables appearing in the source of the conclusion, and*
2. *all variables that appear in the target of a premise where all variables in the source of that premise are source dependent.*

For a source dependent variable v, let $\mathcal{R}$ be the collection of transition relations appearing in a set of premises needed to show source dependency through condition 2. We say that v is source dependent via *the relations in $\mathcal{R}$.*

Note that for a source dependent variable, the set $\mathcal{R}$ is not necessarily unique. For example, in the rule

$$\frac{y \xrightarrow{l_1} y' \quad x \xrightarrow{l_2} z \quad z \xrightarrow{l_3} y'}{f(x,y) \xrightarrow{l} y'}$$

the variable y' is source dependent both via the set $\{\xrightarrow{l_1}\}$ as well as $\{\xrightarrow{l_2}, \xrightarrow{l_3}\}$.

The second auxiliary definition needed for our determinism format is the definition of determinism-respecting substitutions.

Definition 8 (Determinism-Respecting Pairs of Substitutions). *Given a set L of labels, a pair of substitutions (σ, σ') is determinism-respecting w.r.t. a pair of sets of formulae (Φ, Φ') and L when for all two positive formulae $s \xrightarrow{l} s' \in \Phi$ and $t \xrightarrow{l} t' \in \Phi'$ such that $l \in L$, $\sigma(s) \equiv \sigma'(t)$ only if $\sigma(s') \equiv \sigma'(t')$.*

Definition 9 (Determinism Format). *A TSS $\mathcal{T}$ is in the determinism format w.r.t. a set of labels L, when for each $l \in L$ the following conditions hold.*

1. *In each deduction rule $\frac{\Phi}{t \xrightarrow{l} t'}$, each variable $v \in vars(t')$ is source dependent via a subset of $\{\xrightarrow{l} \mid l \in L\}$, and*
2. *for each pair of distinct deduction rules $\frac{\Phi_0}{t_0 \xrightarrow{l} t'_0}$ and $\frac{\Phi_1}{t_1 \xrightarrow{l} t'_1}$ and for each determinism-respecting pair of substitutions (σ, σ') w.r.t. (Φ_0, Φ_1) and L such that $\sigma(t_0) \equiv \sigma'(t_1)$, it holds that either $\sigma(t'_0) \equiv \sigma'(t'_1)$ or $\sigma(\Phi_0)$ contradicts $\sigma'(\Phi_1)$.*

The first constraint in the definition above ensures that each rule in a TSS in the determinism format, with some $l \in L$ as the label of its conclusion, can be used to prove at most one outgoing transition for each closed term. The second requirement guarantees that no two different rules can be used to prove two distinct l-labelled transitions for any closed term.

Theorem 1. *Consider a TSS with (C, U) as its least three-valued stable model and a subset L of its labels. If the TSS is in the determinism format w.r.t. L, then C is deterministic for each $l \in L$.*

For a TSS in the determinism format with (C, U) as its least three-valued stable model, U and thus $C \cup U$ need not be deterministic. The following counterexample illustrates this phenomenon.

Example 1. Consider the TSS given by the following deduction rules.

$$\frac{a \xrightarrow{l} a}{a \xrightarrow{l} b} \qquad \frac{a \overset{l}{\nrightarrow}}{a \xrightarrow{l} a}$$

The above-given TSS is in the determinism format since $a \xrightarrow{l} a$ and $a \overset{l}{\nrightarrow}$ contradict each other (under any substitution). Its least three-valued stable model is, however, $(\emptyset, \{a \xrightarrow{l} a, a \xrightarrow{l} b\})$ and $\{a \xrightarrow{l} a, a \xrightarrow{l} b\}$ is not deterministic.

Example 2. The constraints in Definition 9 are not necessary to ensure determinism. For example, consider the TSS with constant a and rule $x \xrightarrow{a} y$. The transition relation $\xrightarrow{a}$ is obviously deterministic, but the variable y is not source dependent in the rule $x \xrightarrow{a} y$. However, as the following two examples show, relaxing the constraints in Definition 9 may jeopardize determinism.

To see the need for constraint 1, consider the TSS with constant 0 and unary function symbol f with rule $f(x) \xrightarrow{a} y$. This TSS satisfies constraint 2 in Definition 9 vacuously, but the transition relation $\xrightarrow{a}$ it determines is not deterministic since, for instance, $f(0) \xrightarrow{a} p$ holds for each closed term p. Note that the variable y is not source dependent in $f(x) \xrightarrow{a} y$.

The need for constraint 2 is exemplified by the classic non-deterministic choice operator discussed in Example 7. The rules for this operator satisfy constraint 1, but not constraint 2. The transition relations defined by those rules are non-deterministic except for trivial TSSs including this operator.

Corollary 1. *Consider a complete TSS with L as a subset of its labels. If the TSS is in the determinism format w.r.t. L, then its defined transition relation is deterministic for each $l \in L$.*

Constraint 2 in Definition 9 may seem difficult to verify, since it requires checks for all possible (determinism-respecting) substitutions. However, in practical cases, to be quoted in the remainder of this paper, variable names are chosen in such a way that constraint 2 can be checked syntactically. For example, consider the following two deduction rules.

$$\frac{x \xrightarrow{a} x'}{f(x,y) \xrightarrow{a} x'} \qquad \frac{y \overset{a}{\nrightarrow} \quad x \xrightarrow{b} x'}{f(y,x) \xrightarrow{a} x'}$$

If in both deduction rules $f(x,y)$ (or symmetrically $f(y,x)$) was used, it could have been easily seen from the syntax of the rules that the premises of one deduction rule always (under all pairs of substitutions agreeing on the value of x) contradict the premises of the other deduction rule and, hence, constraint 2 is trivially satisfied. Based on this observation, we next present a rule format, whose constraints have a purely syntactic form, and that is sufficiently powerful to handle all the examples we discuss in Section 3.1. (Note that, for the examples in Section 3.1, checking the constraints of Definition 9 is not too hard either.)

Definition 10 (Normalized TSSs). *A TSS is normalized w.r.t. L if each deduction rule is f-defining for some function symbol f, and for each label $l \in L$, each function symbol f and each pair of deduction rules of the form*

$$(\mathrm{r})\ \frac{\Phi_r}{f(\overrightarrow{s}) \xrightarrow{l} s'} \qquad (\mathrm{r}')\ \frac{\Phi_{r'}}{f(\overrightarrow{t}) \xrightarrow{l} t'}$$

the following constraints are satisfied:

1. *the sources of the conclusions coincide, i.e., $f(\overrightarrow{s}) \equiv f(\overrightarrow{t})$,*

2. *each variable $v \in vars(s')$ (symmetrically $v \in vars(t')$) is source dependent in* (r) *(respectively in* (r′)*) via some subset of* $\{\xrightarrow{l} \mid l \in L\}$,
3. *for each variable $v \in vars(r) \cap vars(r')$ there is a set of formulae in $\Phi_r \cap \Phi_{r'}$ proving its source dependency (both in* (r) *and* (r′)*)) via some subset of* $\{\xrightarrow{l} \mid l \in L\}$.

The second and third constraint in Definition 11 guarantee that the syntactic equivalence of relevant terms (the target of the conclusion or the premises negating each other) will lead to syntactically equivalent closed terms under all determinism-respecting pairs of substitutions.

The reader can check that all the examples quoted from the literature in Section 3.1 are indeed normalized TSSs.

Definition 11 (Syntactic Determinism Format). *A normalized TSS is in the (syntactic) determinism format w.r.t. L, when for each two deduction rules* $\frac{\Phi_0}{f(\overrightarrow{s}) \xrightarrow{l} s'}$ *and* $\frac{\Phi_1}{f(\overrightarrow{s}) \xrightarrow{l} s''}$, *it holds that $s' \equiv s''$ or Φ_0 contradicts Φ_1.*

The following theorem states that for normalized TSSs, Definition 11 implies Definition 9.

Theorem 2. *Each normalized TSS in the syntactic determinism format w.r.t. L is also in the determinism format w.r.t. L.*

For brevity, we omit the proof of Theorem 2. The following statement is thus a corollary to Theorems 2 and 1.

Corollary 2. *Consider a normalized TSS with (C, U) as its least three-valued stable model and a subset L of its labels. If the TSS is in the (syntactic) determinism format w.r.t. L (according to Definition 11), then C is deterministic w.r.t. any $l \in L$.*

3.1 Examples

In this section, we present some examples of various TSSs from the literature and apply our (syntactic) determinism format to them. Some of the examples we discuss below are based on TSSs with predicates. The extension of our formats with predicates is straightforward and we discuss it in Section 4.3 to follow.

Example 3 (Conjunctive Nondeterministic Processes). Hennessy and Plotkin, in [10], define a language, called conjunctive nondeterministic processes, for studying logical characterizations of processes. The signature of the language consists of a constant 0, a unary action prefixing operator $a.\_$ for each $a \in A$, and a binary conjunctive nondeterminism operator $\vee$. The operational semantics of this language is defined by the following deduction rules.

$$\frac{}{0 \ \ \mathrm{can}_a} \qquad \frac{}{a.x \ \ \mathrm{can}_a} \qquad \frac{x \ \ \mathrm{can}_a}{x \vee y \ \ \mathrm{can}_a} \qquad \frac{y \ \ \mathrm{can}_a}{x \vee y \ \ \mathrm{can}_a}$$

$$\frac{}{0\ \ \mathrm{after}_a\ \ 0} \qquad \frac{}{a.x\ \ \mathrm{after}_a\ \ x} \qquad \frac{}{a.x\ \ \mathrm{after}_b\ \ 0}\ a \neq b \qquad \frac{x\ \ \mathrm{after}_a\ \ x' \quad y\ \ \mathrm{after}_a\ \ y'}{x \vee y\ \ \mathrm{after}_a\ \ x' \vee y'}$$

The above TSS is in the (syntactic) determinism format with respect to the transition relation after_a (for each $a \in A$). Hence, we can conclude that the transition relations after_a are deterministic.

Example 4 (Delayed Choice). The second example we discuss is a subset of the process algebra $\mathrm{BPA}_{\delta\epsilon} + \mathrm{DC}$ [4], i.e., Basic Process Algebra with deadlock and empty process extended with delayed choice. First we restrict attention to the fragment of this process algebra without non-deterministic choice $+$ and with action prefix $a.\_$ instead of general sequential composition $\cdot$. This altered process algebra has the following deduction rules, where a ranges over the set of actions A:

$$\frac{}{\epsilon\downarrow} \qquad \frac{}{a.x \xrightarrow{a} x} \qquad \frac{x\downarrow}{x \mp y\downarrow} \qquad \frac{y\downarrow}{x \mp y\downarrow}$$

$$\frac{x \xrightarrow{a} x' \qquad y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} x' \mp y'} \qquad \frac{x \xrightarrow{a} x' \qquad y \overset{a}{\not\rightarrow}}{x \mp y \xrightarrow{a} x'} \qquad \frac{x \overset{a}{\not\rightarrow} \qquad y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} y'}$$

In the above specification, predicate $p\downarrow$ denotes the possibility of termination for process p. The intuition behind the delayed choice operator, denoted by $\_ \mp \_$, is that the choice between two components is only resolved when one performs an action that the other cannot perform. When both components can perform an action, the delayed choice between them remains unresolved and the two components synchronize on the common action. This transition system specification is in the (syntactic) determinism format w.r.t. $\{a \mid a \in A\}$.

Addition of non-deterministic choice $+$ or sequential composition $\cdot$ results in deduction rules that do not satisfy the determinism format. For example, addition of sequential composition comes with the following deduction rules:

$$\frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \qquad \frac{x\downarrow \quad y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'}$$

The sets of premises of these rules do not contradict each other. The extended TSS is indeed non-deterministic since, for example, $(\epsilon \mp (a.\epsilon)) \cdot (a.\epsilon) \xrightarrow{a} \epsilon$ and $(\epsilon \mp (a.\epsilon)) \cdot (a.\epsilon) \xrightarrow{a} \epsilon \cdot (a.\epsilon)$.

Example 5 (Time Transitions I). This example deals with the Algebra of Timed Processes, ATP, of Nicollin and Sifakis [17]. In the TSS given below, we specify the time transitions (denoted by label χ) of delayable deadlock δ, non-deterministic choice $\_ \oplus \_$, unit-delay operator $\lfloor\_\rfloor\_$ and parallel composition $\_ \parallel \_$.

$$\frac{}{\delta \xrightarrow{\chi} \delta} \qquad \frac{x \xrightarrow{\chi} x' \quad y \xrightarrow{\chi} y'}{x \oplus y \xrightarrow{\chi} x' \oplus y'} \qquad \frac{}{\lfloor x \rfloor (y) \xrightarrow{\chi} y} \qquad \frac{x \xrightarrow{\chi} x' \quad y \xrightarrow{\chi} y'}{x \parallel y \xrightarrow{\chi} x' \parallel y'}$$

These deduction rules all trivially satisfy the determinism format for time transitions since the sources of conclusions of different deduction rules cannot be unified. Also the additional operators involving time, namely, the delay operator $\lfloor \_ \rfloor^{d} \_$, execution delay operator $\lceil \_ \rceil^{d} \_$ and unbounded start delay operator $\lfloor \_ \rfloor^{\omega}$, satisfy the determinism format for time transitions. The deduction rules are given below, for $d \geq 1$:

$$\frac{}{\lfloor x \rfloor^{1}(y) \xrightarrow{\chi} y} \qquad \frac{x \xrightarrow{\chi} x'}{\lfloor x \rfloor^{d+1}(y) \xrightarrow{\chi} \lfloor x' \rfloor^{d}(y)} \qquad \frac{x \not\xrightarrow{\chi}}{\lfloor x \rfloor^{d+1}(y) \xrightarrow{\chi} \lfloor x \rfloor^{d}(y)}$$

$$\frac{x \xrightarrow{\chi} x'}{\lfloor x \rfloor^{\omega} \xrightarrow{\chi} \lfloor x' \rfloor^{\omega}} \qquad \frac{x \not\xrightarrow{\chi}}{\lfloor x \rfloor^{\omega} \xrightarrow{\chi} \lfloor x \rfloor^{\omega}}$$

$$\frac{x \xrightarrow{\chi} x'}{\lceil x \rceil^{1}(y) \xrightarrow{\chi} y} \qquad \frac{x \xrightarrow{\chi} x'}{\lceil x \rceil^{d+1}(y) \xrightarrow{\chi} \lceil x' \rceil^{d}(y)}$$

Example 6 (Time Transitions II). Most of the timed process algebras that originate from the Algebra of Communicating Processes (ACP) from [5,3] such as those reported in [2] have a deterministic time transition relation as well.

In the TSS given below, the time unit delay operator is denoted by $\sigma_{\mathrm{rel}}\_$, nondeterministic choice is denoted by $\_ + \_$, and sequential composition is denoted by $\_ \cdot \_$. The deduction rules for the time transition relation for this process algebra are the following:

$$\frac{}{\sigma_{\mathrm{rel}}(x) \xrightarrow{1} x} \qquad \frac{x \xrightarrow{1} x' \quad y \xrightarrow{1} y'}{x + y \xrightarrow{1} x' + y'} \qquad \frac{x \xrightarrow{1} x' \quad y \not\xrightarrow{1}}{x + y \xrightarrow{1} x'} \qquad \frac{x \not\xrightarrow{1} \quad y \xrightarrow{1} y'}{x + y \xrightarrow{1} y'}$$

$$\frac{x \xrightarrow{1} x' \quad x \not\downarrow}{x \cdot y \xrightarrow{1} x' \cdot y} \qquad \frac{x \xrightarrow{1} x' \quad y \not\xrightarrow{1}}{x \cdot y \xrightarrow{1} x' \cdot y} \qquad \frac{x \xrightarrow{1} x' \quad x\downarrow \quad y \xrightarrow{1} y'}{x \cdot y \xrightarrow{1} x' \cdot y + y'} \qquad \frac{x \not\xrightarrow{1} \quad x\downarrow \quad y \xrightarrow{1} y'}{x \cdot y \xrightarrow{1} y'}$$

Note that here we have an example of deduction rules, the first two deduction rules for time transitions of a sequential composition, for which the premises do not contradict each other. Still these deduction rules satisfy the determinism format since the targets of the conclusions are identical. In the syntactically richer framework of [21], where arbitrary first-order logic formulae over transitions are allowed, those two deduction rules are presented by a single rule with premise $x \xrightarrow{1} x' \wedge (x \not\downarrow \vee\, y \not\xrightarrow{1})$.

Sometimes such timed process algebras have an operator for specifying an arbitrary delay, denoted by $\sigma^{*}_{\mathrm{rel}}\_$, with the following deduction rules.

$$\frac{x \not\xrightarrow{1}}{\sigma^{*}_{\mathrm{rel}}(x) \xrightarrow{1} \sigma^{*}_{\mathrm{rel}}(x)} \qquad \frac{x \xrightarrow{1} x'}{\sigma^{*}_{\mathrm{rel}}(x) \xrightarrow{1} x' + \sigma^{*}_{\mathrm{rel}}(x)}$$

The premises of these rules contradict each other and so, the semantics of this operator also satisfies the constraints of our (syntactic) determinism format.

4 Idempotence

Our order of business in this section is to present a rule format that guarantees the idempotence of certain binary operators. In the definition of our rule format, we rely implicitly on the work presented in the previous section.

4.1 Format

Definition 12 (Idempotence). *A binary operator $f \in \Sigma$ is* idempotent w.r.t. an equivalence $\sim$ *on closed terms if and only if for each $p \in \mathbb{C}(\Sigma)$, it holds that $f(p, p) \sim p$.*

Idempotence is defined with respect to a notion of behavioral equivalence. There are various notions of behavioral equivalence defined in the literature, which are, by and large, weaker than bisimilarity defined below. Thus, to be as general as possible, we prove our idempotence result for all notions that contain, i.e., are weaker than, bisimilarity.

Definition 13 (Bisimulation). *Let $\mathcal{T}$ be a TSS with signature Σ. A relation $\mathcal{R} \subseteq \mathbb{C}(\Sigma) \times \mathbb{C}(\Sigma)$ is a* bisimulation relation *if and only if $\mathcal{R}$ is symmetric and for all $p_0, p_1, p'_0 \in \mathbb{C}(\Sigma)$ and $l \in L$*

$$(p_0 \mathrel{\mathcal{R}} p_1 \wedge \mathcal{T} \vdash p_0 \xrightarrow{l} p'_0) \Rightarrow \exists_{p'_1 \in \mathbb{C}(\Sigma)} (\mathcal{T} \vdash p_1 \xrightarrow{l} p'_1 \wedge p'_0 \mathrel{\mathcal{R}} p'_1).$$

Two terms $p_0, p_1 \in \mathbb{C}(\Sigma)$ are called bisimilar, *denoted by $p_0 \leftrightarroweq p_1$, when there exists a bisimulation relation $\mathcal{R}$ such that $p_0 \mathcal{R} p_1$.*

Definition 14 (The Idempotence Rule Format). *Let $\gamma : L \times L \to L$ be a partial function such that $\gamma(l_0, l_1) \in \{l_0, l_1\}$ if it is defined. We define the following two rule forms.*

1_l. *Choice rules*

$$\frac{\{x_i \xrightarrow{l} t\} \cup \Phi}{f(x_0, x_1) \xrightarrow{l} t}, \quad i \in \{0, 1\}$$

$2_{l_0,l_1}$. *Communication rules*

$$\frac{\{x_0 \xrightarrow{l_0} t_0,\ x_1 \xrightarrow{l_1} t_1\} \cup \Phi}{f(x_0, x_1) \xrightarrow{\gamma(l_0,l_1)} f(t_0, t_1)}, \quad t_0 \equiv t_1 \text{ or } (l_0 = l_1 \text{ and } \xrightarrow{l_0} \text{ is deterministic })$$

In each case, Φ can be an arbitrary, possibly empty set of (positive or negative) formulae.

In addition, we define the starred version of each form, 1^*_l and $2^*_{l_0,l_1}$. The starred version of each rule is the same as the unstarred one except that t, t_0 and t_1 are restricted to be concrete variables and the set Φ must be empty in each case.

A TSS is in idempotence format w.r.t. a binary operator f *if each f-defining rule, i.e., a deduction rule with f appearing in the source of the conclusion, is of the forms 1_l or $2_{l_0,l_1}$, for some $l, l_0, l_1 \in L$, and for each label $l \in L$ there exists at least one rule of the forms 1^*_l or $2^*_{l,l}$.*

We should note that the starred versions of the forms are special cases of their unstarred counterparts; for example a rule which has form 1^*_l also has form 1_l.

Theorem 3. *Assume that a TSS is complete and is in the idempotence format with respect to a binary operator f. Then, f is idempotent w.r.t. to any equivalence $\sim$ such that $\underline{\leftrightarrow} \subseteq \sim$.*

4.2 Relaxing the Restrictions

In this section we consider the constraints of the idempotence rule format and show that they cannot be dropped without jeopardizing the meta-theorem.

First of all we note that, in rule form 1_l, it is necessary that the label of the premise matches the label of the conclusion. If it does not, in general, we cannot prove that $f(p,p)$ simulates p or vice versa. This requirement can be stated more generally for both rule forms in Definition 14; the label of the conclusion must be among the labels of the premises. The requirement that $\gamma(l,l') \in \{l,l'\}$ exists to ensure this constraint for form $2_{l,l'}$. A simple synchronization rule provides a counter-example that shows why this restriction is needed. Consider the following TSS with constants 0, τ, a and $\bar{a}$ and two binary operators $+$ and $\|$:

$$\frac{}{\alpha \xrightarrow{\alpha} 0} \qquad \frac{x \xrightarrow{\alpha} x'}{x + y \xrightarrow{\alpha} x'} \qquad \frac{y \xrightarrow{\alpha} y'}{x + y \xrightarrow{\alpha} y'} \qquad \frac{x \xrightarrow{a} x' \quad y \xrightarrow{\bar{a}} y'}{x \parallel y \xrightarrow{\tau} x' \parallel y'}$$

where α is τ, a or $\bar{a}$. Here it is easy to see that although $(a+\bar{a}) \parallel (a+\bar{a})$ has an outgoing τ-transition, $a+\bar{a}$ does not afford such a transition.

The condition that for each l at least one rule of the forms 1^*_l or $2^*_{l,l}$ must exist comprises a few constraints on the rule format. First of all, it says there must be at least one f-defining rule. If not, it is easy to see that there could exist a process p where $f(p,p)$ deadlocks (since there are no f-defining rules) but p does not. It also states that there must be at least one rule in the starred form, where the targets are restricted to variables. To motivate these constraints, consider the following TSS.

$$\frac{}{a \xrightarrow{a} 0} \qquad \frac{x \xrightarrow{a} a}{f(x,y) \xrightarrow{a} a}$$

The processes a and $f(a,a)$ are not bisimilar as the former can do an a-transition but the latter is stuck. The starred forms also require that Φ is empty, i.e. there is no testing. This is necessary in the proof of Theorem 3 because in the presence of extra premises, we cannot in general instantiate such a rule to show that $f(p,p)$ simulates p. Finally, the condition requires that if we rely on a rule of the form $2^*_{l,l'}$ and $t_0 \not\equiv t_1$, then the labels l and l' in the premises of the rule must coincide.

To see why, consider a TSS containing a *left synchronize* operator $\rrbracket$, one that synchronizes a step from each operand but uses the label of the left one. Here we let $\alpha \in \{a, \bar{a}\}$.

$$\frac{}{\alpha \xrightarrow{\alpha} 0} \quad \frac{x \xrightarrow{\alpha} x'}{x + y \xrightarrow{\alpha} x'} \quad \frac{y \xrightarrow{\alpha} y'}{x + y \xrightarrow{\alpha} y'} \quad \frac{x \xrightarrow{a} x' \quad y \xrightarrow{\bar{a}} y'}{x \rrbracket y \xrightarrow{a} x' \rrbracket y'}$$

In this TSS the processes $(a + \bar{a})$ and $(a + \bar{a}) \rrbracket (a + \bar{a})$ are not bisimilar since the latter does not afford an $\bar{a}$-transition whereas the former does.

For rules of form $2_{l,l'}$ we require that either $t_0 \equiv t_1$, or that the mentioned labels are the same and the associated transition relation is deterministic. This requirement is necessary in the proof of Theorem 3 to ensure that the target of the conclusion fits our definition of $\simeq_f$, i.e. the operator is applied to two identical terms. Consider the following TSS where $\alpha \in \{a, b\}$.

$$\frac{}{a \xrightarrow{a} a} \quad \frac{}{a \xrightarrow{a} b} \quad \frac{}{b \xrightarrow{b} b} \quad \frac{x \xrightarrow{\alpha} x' \quad y \xrightarrow{\alpha} y'}{x \mid y \xrightarrow{\alpha} x' \mid y'}$$

For the operator $|$, this violates the condition $t_0 \equiv t_1$ (note that $\xrightarrow{a}$ is not deterministic). We observe that $a|a \xrightarrow{a} a|b$. The only possibilities for a to simulate this a-transition is either with $a \xrightarrow{a} a$ or with $a \xrightarrow{a} b$. However, neither a nor b can be bisimilar to $a \mid b$ because both a and b have outgoing transitions while $a \mid b$ is stuck. Therefore a and $a \mid a$ cannot be bisimilar. If $t_0 \not\equiv t_1$ we must require that the labels match, $l_0 = l_1$, and that $\xrightarrow{l_0}$ is deterministic. We require the labels to match because if they do not, then given only $p \xrightarrow{l} p'$ it is impossible to prove that $f(p, p)$ can simulate it using only a $2^*_{l,l'}$ rule. The determinacy of the transition with that label is necessary when proving that transitions from $f(p, p)$ can, in general, be simulated by p; if we assume that $f(p, p) \xrightarrow{l} p'$ then we must be able to conclude that p' has the shape $f(p'', p'')$ for some p'', in order to meet the bisimulation condition for $\simeq_f$. Consider the standard choice operator $+$ and prefixing operator . of CCS with the $|$ operator from the last example, with $\alpha \in \{a, b, c\}$.

$$\frac{}{\alpha \xrightarrow{\alpha} 0} \quad \frac{}{\alpha.x \xrightarrow{\alpha} x} \quad \frac{x \xrightarrow{\alpha} x'}{x + y \xrightarrow{\alpha} x'} \quad \frac{y \xrightarrow{\alpha} y'}{x + y \xrightarrow{\alpha} y'} \quad \frac{x \xrightarrow{\alpha} x' \quad y \xrightarrow{\alpha} y'}{x \mid y \xrightarrow{\alpha} x' \mid y'}$$

If we let $p = a.b + a.c$, then $p \mid p \xrightarrow{a} b \mid c$ and $b \mid c$ is stuck. However, p cannot simulate this transition w.r.t. $\simeq_f$. Indeed, p and $p \mid p$ are not bisimilar.

4.3 Predicates

There are many examples of TSSs where predicates are used. The definitions presented in Section 2 and 4 can be easily adapted to deal with predicates as well. In particular, two closed terms are called bisimilar in this setting when, in addition to the transfer conditions of bisimilarity, they satisfy the same predicates. To extend the idempotence rule format to a setting with predicates, the following types of rules for predicates are introduced:

3_P. *Choice rules for predicates*

$$\frac{\{Px_i\} \cup \Phi}{Pf(x_0, x_1)}, \quad i \in \{0, 1\}$$

4_P. *Synchronization rules for predicates*

$$\frac{\{Px_0,\, Px_1\} \cup \Phi}{Pf(x_0, x_1)}$$

As before, we define the starred version of these forms, 3^*_P and 4^*_P. The starred version of each rule is the same as the unstarred one except that the set Φ must be empty in each case. With these additional definitions the idempotence format is defined as follows.

A TSS is in *idempotence format w.r.t. a binary operator* f if each f-defining rule, i.e., a deduction rule with f appearing in the source of the conclusion, is of one the forms 1_l, $2_{l_0,l_1}$, 3_P or 4_P for some $l, l_0, l_1 \in L$, and predicate symbol P. Moreover, for each $l \in L$, there exists at least one rule of the forms 1^*_l or $2^*_{l,l}$, and for each predicate symbol P there is a rule of the form 3^*_P or 4^*_P.

4.4 Examples

Example 7. The most prominent example of an idempotent operator is non-deterministic choice, denoted +. It typically has the following deduction rules:

$$\frac{x_0 \xrightarrow{a} x'_0}{x_0 + x_1 \xrightarrow{a} x'_0} \qquad \frac{x_1 \xrightarrow{a} x'_1}{x_0 + x_1 \xrightarrow{a} x'_1}$$

Clearly, these are in the idempotence format w.r.t. +.

Example 8 (External Choice). The well-known external choice operator $\Box$ from CSP [11] has the following deduction rules

$$\frac{x_0 \xrightarrow{a} x'_0}{x_0 \,\Box\, x_1 \xrightarrow{a} x'_0} \qquad \frac{x_1 \xrightarrow{a} x'_1}{x_0 \,\Box\, x_1 \xrightarrow{a} x'_1} \qquad \frac{x_0 \xrightarrow{\tau} x'_0}{x_0 \,\Box\, x_1 \xrightarrow{\tau} x'_0 \,\Box\, x_1} \qquad \frac{x_1 \xrightarrow{\tau} x'_1}{x_0 \,\Box\, x_1 \xrightarrow{\tau} x_0 \,\Box\, x'_1}$$

Note that the third and fourth deduction rule are not instances of any of the allowed types of deduction rules. Therefore, no conclusion about the validity of idempotence can be drawn from our format. In this case this does not point to a limitation of our format, because this operator is not idempotent in strong bisimulation semantics [7].

Example 9 (Strong Time-Deterministic Choice). The choice operator that is used in the timed process algebra ATP [17] has the following deduction rules.

$$\frac{x_0 \xrightarrow{a} x'_0}{x_0 \oplus x_1 \xrightarrow{a} x'_0} \qquad \frac{x_1 \xrightarrow{a} x'_1}{x_0 \oplus x_1 \xrightarrow{a} x'_1} \qquad \frac{x_0 \xrightarrow{\chi} x'_0 \quad x_1 \xrightarrow{\chi} x'_1}{x_0 \oplus x_1 \xrightarrow{\chi} x'_0 \oplus x'_1}$$

The idempotence of this operator follows from our format since the last rule for $\oplus$ fits the form $2^*_{\chi,\chi}$ because, as we remarked in Example 5, the transition relation $\xrightarrow{\chi}$ is deterministic.

Example 10 (Weak Time-Deterministic Choice). The choice operator + that is used in most ACP-style timed process algebras has the following deduction rules.

$$\frac{x_0 \xrightarrow{a} x_0'}{x_0 + x_1 \xrightarrow{a} x_0'} \qquad \frac{x_1 \xrightarrow{a} x_1'}{x_0 + x_1 \xrightarrow{a} x_1'}$$

$$\frac{x_0 \xrightarrow{1} x_0' \quad x_1 \xrightarrow{1} x_1'}{x_0 + x_1 \xrightarrow{1} x_0' + x_1'} \qquad \frac{x_0 \xrightarrow{1} x_0' \quad x_1 \stackrel{1}{\not\rightarrow}}{x_0 + x_1 \xrightarrow{1} x_0'} \qquad \frac{x_0 \stackrel{1}{\not\rightarrow} \quad x_1 \xrightarrow{1} x_1'}{x_0 + x_1 \xrightarrow{1} x_1'}$$

The third deduction rule is of the form $2^*_{1,1}$, the others are of forms 1^*_a and 1^*_1. This operator is idempotent (since the transition relation $\xrightarrow{1}$ is deterministic, as remarked in Example 6).

Example 11 (Conjunctive Nondeterminism). The operator $\vee$ as defined in Example 3 by means of the deduction rules

$$\frac{x \ \ \mathrm{can}_a}{x \vee y \ \ \mathrm{can}_a} \qquad \frac{y \ \ \mathrm{can}_a}{x \vee y \ \ \mathrm{can}_a} \qquad \frac{x \ \ \mathrm{after}_a \ x' \quad y \ \ \mathrm{after}_a \ y'}{x \vee y \ \ \mathrm{after}_a \ x' \vee y'}$$

satisfies the idempotence format (extended to a setting with predicates). The first two deduction rules are of the form $3^*{}_{\mathrm{can}_a}$ and the last one is of the form $2^*_{a,a}$. Here we have used the fact that the transition relations after_a are deterministic as concluded in Example 3.

Example 12 (Delayed Choice). Delayed choice can be concluded to be idempotent in the restricted setting without + and $\cdot$ by using the idempotence format and the fact that in this restricted setting the transition relations $\xrightarrow{a}$ are deterministic. (See Example 4.)

$$\frac{x\downarrow}{x \mp y\downarrow} \qquad \frac{y\downarrow}{x \mp y\downarrow} \qquad \frac{x \xrightarrow{a} x' \quad y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} x' \mp y'} \qquad \frac{x \xrightarrow{a} x' \quad y \stackrel{a}{\not\rightarrow}}{x \mp y \xrightarrow{a} x'} \qquad \frac{x \stackrel{a}{\not\rightarrow} \quad y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} y'}$$

The first two deduction rules are of form $3^*_\downarrow$, the third one is a $2^*_{a,a}$ rule, and the others are 1_a rules. Note that for any label a starred rule is present.

For the extensions discussed in Example 4 idempotence cannot be established using our rule format since the transition relations are no longer deterministic. In fact, delayed choice is not idempotent in those cases.

5 Conclusions

In this paper, we presented two rule formats guaranteeing determinism of certain transitions and idempotence of binary operators. Our rule formats cover all practical cases of determinism and idempotence that we have thus far encountered in the literature.

We plan to extend our rule formats with the addition of data/store. Also, it is interesting to study the addition of structural congruences pertaining to idempotence to the TSSs in our idempotence format.

References

1. Aceto, L., Fokkink, W.J., Verhoef, C.: Structural operational semantics. In: Handbook of Process Algebra, ch. 3, pp. 197–292. Elsevier, Amsterdam (2001)
2. Baeten, J.C.M., Middelburg, C.A.: Process Algebra with Timing. EATCS Monographs. Springer, Berlin (2002)
3. Baeten, J.C.M., Weijland, W.P.: Process Algebra. Cambridge Tracts in Theoretical Computer Science, vol. 18. Cambridge University Press, Cambridge (1990)
4. Baeten, J.C.M., Mauw, S.: Delayed choice: An operator for joining Message Sequence Charts. In: Proceedings of Formal Description Techniques. IFIP Conference Proceedings, vol. 6, pp. 340–354. Chapman & Hall, Boca Raton (1995)
5. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control 60(1-3), 109–137 (1984)
6. Cranen, S., Mousavi, M.R., Reniers, M.A.: A rule format for associativity. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 447–461. Springer, Heidelberg (2008)
7. D'Argenio, P.R.: τ-angelic choice for process algebras (revised version). Technical report, LIFIA, Depto. de Informática, Fac. de Cs. Exactas, Universidad Nacional de La Plata (1995)
8. Fokkink, W.J., Duong Vu, T.: Structural operational semantics and bounded nondeterminism. Acta Informatica 39(6-7), 501–516 (2003)
9. Groote, J.F.: Transition system specifications with negative premises. Theoretical Computer Science 118(2), 263–299 (1993)
10. Hennessy, M., Plotkin, G.D.: Finite conjuncitve nondeterminism. In: Rozenberg, G. (ed.) APN 1987. LNCS, vol. 266, pp. 233–244. Springer, Heidelberg (1987)
11. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
12. Lanotte, R., Tini, S.: Probabilistic congruence for semistochastic generative processes. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 63–78. Springer, Heidelberg (2005)
13. Mousavi, M.R., Phillips, I.C.C., Reniers, M.A., Ulidowski, I.: Semantics and expressiveness of ordered SOS. Information and Computation 207(2), 85–119 (2009)
14. Mousavi, M.R., Reniers, M.A.: Orthogonal extensions in structural operational semantics. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1214–1225. Springer, Heidelberg (2005)
15. Mousavi, M.R., Reniers, M.A., Groote, J.F.: A syntactic commutativity format for SOS. Information Processing Letters 93, 217–223 (2005)
16. Mousavi, M.R., Reniers, M.A., Groote, J.F.: SOS formats and meta-theory: 20 years after. Theoretical Computer Science (373), 238–272 (2007)
17. Nicollin, X., Sifakis, J.: The algebra of timed processes ATP: Theory and application. Information and Computation 114(1), 131–178 (1994)

18. Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Progamming 60, 17–139 (2004); This article first appeared as Technical Report DAIMI FN-19, Computer Science Department, Aarhus University
19. Tini, S.: Rule formats for compositional non-interference properties. Journal of Logic and Algebraic Progamming 60, 353–400 (2004)
20. Ulidowski, I., Yuen, S.: Process languages with discrete relative time based on the ordered SOS format and rooted eager bisimulation. Journal of Logic and Algebraic Progamming 60, 401–460 (2004)
21. van Weerdenburg, M., Reniers, M.A.: Structural operational semantics with first-order logic. In: Pre-proceedings of SOS 2008, pp. 48–62 (2008)
22. Verhoef, C.: A congruence theorem for structured operational semantics with predicates and negative premises. Nordic Journal of Computing 2(2), 274–302 (1995)

The Complexity of Reachability in Randomized Sabotage Games[⋆]

Dominik Klein, Frank G. Radmacher, and Wolfgang Thomas

Lehrstuhl für Informatik 7, RWTH Aachen University, Germany
dominik.klein@rwth-aachen.de,
radmacher@automata.rwth-aachen.de,
thomas@automata.rwth-aachen.de

Abstract. We analyze a model of fault-tolerant systems in a probabilistic setting. The model has been introduced under the name of "sabotage games". A reachability problem over graphs is considered, where a "Runner" starts from a vertex u and seeks to reach some vertex in a target set F while, after each move, the adversary "Blocker" deletes one edge. Extending work by Löding and Rohde (who showed PSpace-completeness of this reachability problem), we consider the randomized case (a "game against nature") in which the deleted edges are chosen at random, each existing edge with the same probability. In this much weaker model, we show that, for any probability p and $\varepsilon > 0$, the following problem is again PSpace-complete: Given a game graph with u and F and a probability p' in the interval $[p-\varepsilon, p+\varepsilon]$, is there a strategy for Runner to reach F with probability $\geq p'$? Our result extends the PSpace-completeness of Papadimitriou's "dynamic graph reliability"; there, the probabilities of edge failures may depend both on the edge and on the current position of Runner.

Keywords: games, reachability, probabilistic systems, fault-tolerant systems.

1 Introduction

The subject of this paper is a model of fault-tolerant computation in which a reachability objective over a network is confronted with the failure of connections (edges). It is well known that adding dynamics to originally static models makes their analysis much harder – in our case, these dynamics are generated by the feature of vanishing edges in graphs. We build on hardness results of Löding and Rohde that are explained in more detail below. In the present paper we combine the aspect of dynamics with probability assumptions, which makes the model "coarser" or "softer". We show that, even in the probabilistic framework, the hardness phenomena of the standard dynamic model are still valid. Technically

[⋆] This research was supported by the RWTH Aachen Research Cluster UMIC of the German Excellence Initiative, German Research Foundation grant DFG EXC 89.

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 162–177, 2010.

speaking, we show that the results of Löding and Rohde are preserved in the more general randomized framework.

Specifically, we consider a two player game based on the model of (discrete) "sabotage games" suggested by van Benthem (cf. [12]). These games are played on graphs which edges may be multi-edges. A player, called "Runner", moves along edges and wants to reach a vertex in a set F from a given initial vertex u. After each move of Runner, the adversary, called "Blocker", may delete an edge; and in this way Runner and Blocker move in alternation. The algorithmic problem of "solving this game" asks for a winning strategy for Runner that enables him to reach a vertex in F against any choice of Blocker in deleting edges. The theory of these games was developed by Löding and Rohde; see [5,6,4,10,11]. Also, other winning conditions (more general than reachability) were considered; see [5,11,12].

In the present paper, we modify the sabotage games in a way that corresponds to a more realistic modeling: The second player Blocker is replaced by random fault. In each turn an edge (which may be a multi-edge) between two nodes is hit – each existing (multi-) edge with the same probability – and its multiplicity is reduced (resp. deleted in case of a single edge). So, in this approach, the player Runner is not faced with a Blocker, but rather has to play against "nature" [7]. There are several scenarios that motivate this model, e.g. the "Traveling Researcher Problem" as formulated by van Benthem [12], or the analysis of routing problems in networks where routing is subject to change, reacting to failures of connections. In such cases, it is rarely realistic to assume that there is an omniscient adversary that manipulates the world. The faults in natural scenarios are usually better modeled as random events. In our work, we use the term "randomized sabotage game" to emphasize our starting point, the sabotage games; but one might prefer to speak of reachability games against nature.

Our studies extend previous results in two ways: On the one hand, this paper extends the classical framework of sabotage games in which Löding and Rohde showed the PSpace-completeness of solving sabotage games with reachability winning conditions [5]. The natural question arises whether this result can be transferred when replacing the adversary player Blocker by arbitrary fault. On the other hand, our work is closely related to a similar question which was studied by Papadimitriou in his work on "games against nature" [7]. He considered the problem of "dynamic graph reliability" (DGR), where each edge e fails with a probability $p(e, v)$ after each turn, i.e. the probability depends on both the edge e and the current position v of Runner. Papadimitriou's game model against nature is rather strong, since for all edges the probabilities of deletion can be adjusted after each turn; his proof for the PSpace-hardness of DGR heavily depends on these adjustments. In fact, all problems that are considered in [7,8] as "games against nature" allow a precise adjustment of the probability for arbitrary large games, so that a reduction from the PSpace-complete problem *SSAT* [7,3] is possible (which is a stochastic version of SAT, with radmoized quantifiers instead of universal quantifiers). However, it should be noted that randomized sabotage games are not a special case of DGR, since, in randomized sabotage games, exactly one edge is deleted in each turn. In this paper, we pursue the question of whether Papadimitriou's result can be extended to a game model with a uniform probability distribution (e.g. in each turn, one of the n edges is deleted with probability $p = \frac{1}{n}$).

Our main result says that, in randomized sabotage games with a uniform distribution of failures where exactly one edge is deleted per turn, for any $p \in [0, 1]$ and $\varepsilon > 0$ the following problem is PSpace-complete: Given a game arena with origin u, a distinguished set F, and a probability p' in the interval $[p - \varepsilon, p + \varepsilon]$, does Runner have a strategy to reach F from u with probability $\geq p'$?

The remainder of this paper is structured as follows. In Section 2 we introduce the basic notions of randomized sabotage games. Section 3 is concerned with the PSpace-hardness of the reachability version of randomized sabotage games. Here we start from the construction of Löding and Rohde [5] on PSpace-hardness for the original sabotage game. For infinitely many probabilities $p_{k,n} \in [0, 1]$ we reduce the PSpace-complete problem of *Quantified Boolean Formulae* (QBF) (see [8], problem "QSAT") to the question of whether, given a randomized sabotage game with u and F, the goal set F is reachable with a probability of $p_{k,n}$. In order to complete the proof of our main result, we show in Section 4 that the set of the probabilities $p_{k,n}$ is dense in the interval $[0, 1]$, and that the parameters k and n can be computed efficiently such that $p_{k,n} \in [p - \varepsilon, p + \varepsilon]$. In Section 5 we address perspectives which are treated in ongoing work.

2 The Randomized Sabotage Game

A sabotage game is played on a graph (V, E), where V is a set of vertices. The vertices are connected by a set of edges, given by $E \subseteq V \times V$. We will assume undirected graphs from now on, i.e. $(u, v) \in E \Rightarrow (v, u) \in E$; however, the ideas presented here also work for directed graphs in the same way. It should also be noted that, while in the "classical" notion of sabotage games multi-edges are allowed, we will restrict ourselves to single edges only. Clearly, the hardness result presented here also holds for the case of multi-edges (and also, the problem still belongs to PSpace).

A *randomized sabotage game* is a tuple $\mathcal{G} = (G, v_{\text{in}})$ with a graph $G = (V, E_{\text{in}})$ and the initial vertex $v_{\text{in}} \in V$. A position of the game is a tuple (v_i, E_i). The initial position is $(v_{\text{in}}, E_{\text{in}})$. In each turn of the game, the player – called Runner – chooses an outgoing edge (v_n, v_{n+1}) in vertex v_n of position (v_n, E_n) and moves to vertex v_{n+1}. Then, a dice with $|E_n|$ sides is thrown and the chosen edge e is removed from E_n. We define $E_{n+1} := E_n \setminus \{e\}$. After this turn, the new position of the game is (v_{n+1}, E_{n+1}); we say that Runner has *reached* the vertex v_{n+1}.

Clearly, since edges are only deleted and not added, each play and the number of positions are finite. We only consider *reachability* as winning condition in this paper: For a randomized sabotage game $\mathcal{G} = ((V, E_{\text{in}}), v_{\text{in}})$ with a set of *final vertices* $F \subseteq V$, Runner wins a play iff he reaches a final vertex $v \in F$.

For the probabilistic analysis, we build up the game tree $t_{\mathcal{G}}$ for any randomized sabotage game $\mathcal{G}$. It is convenient to introduce tree nodes also for the intermediate positions that result from nature's moves, i.e. from edge deletions (in the following nature's positions are marked with an overline). The root of the game tree is $(v_{\text{in}}, E_{\text{in}})$, where Runner starts to move. From a position (v, E) with $v \notin F$ where Runner moves, the successor nodes are all positions $\overline{(v', E)}$ with $(v, v') \in E$ (a position (v, E), with $v \in F$ or $(v, v') \notin E$ for all v', is a leaf). Now, the successors of $\overline{(v', E)}$ are the positions (v', E') where E' results from E by an edge deletion.

To each node of Runner we associate the probability for Runner to win the subgame starting in this node. This probability is computed inductively in the obvious way, starting with 1 and 0 at a leaf (v, E) depending on whether $v \in F$ or not. For an inner node s of Runner with successors s_i of Nature, suppose that s_i has k successors $s_{i1}, \ldots, s_{ik}$ (where again Runner moves) which have, by induction, probabilities p_{ij} for Runner to win. We associate to each s_i the probability $p_i := \frac{1}{k} \sum_j p_{ij}$; then we pick the maximal p_i that occurs and associate it to s (a node with this maximal probability will be chosen by Runner). We say that Runner wins with probability p if the root of the game tree has a probability p.

Let p be an arbitrary number in $[0, 1]$. The Problem *Randomized Reachability Game for probability* p is the following:

> Given a randomized sabotage game $\mathcal{G} = ((V, E_{\text{in}}), v_{\text{in}})$ and a designated set $F \subseteq V$, does Runner win this game with probability $\geq p$?

Löding and Rohde have already shown that solving classical sabotage games with reachability winning condition is PSpace-complete [5]. So, the randomized sabotage game problem for probability $p = 1$ is PSpace-hard. On the other hand, the problem of whether Runner wins a randomized sabotage game with a probability $p > 0$ is decidable in linear time, because Runner wins with a probability > 0 iff there is a path from the initial to a final vertex.

Our main result says that the problem remains PSpace-hard if we restrict the probability to any interval: For any fixed $p \in [0, 1]$ and $\varepsilon > 0$, the randomized reachability game problem for a probability p' which may vary in the interval $[p - \varepsilon, p + \varepsilon]$ is PSpace-complete. More precisely:

Theorem 2.1. *For each fixed* $p \in [0, 1]$ *and* $\varepsilon > 0$*, the following is* PSpace*-complete: Given a randomized sabotage game* $\mathcal{G}$ *with goal set* F *and a probability* $p' \in [p - \varepsilon, p + \varepsilon]$*, does Runner win* $\mathcal{G}$ *with probability* $\geq p'$*?*

For the proof we use a parametrized reduction of the problem *Quantified Boolean Formulae* (QBF): For each QBF-sentence φ, we construct a family of instances $\mathcal{G}_{\varphi,k,n}$ and $p_{k,n}$ such that, for each k and n, the sentence φ is true iff, over $\mathcal{G}_{\varphi,k,n}$ with u and F, Runner wins with probability $\geq p_{k,n}$. Furthermore, we guarantee that, given p, $\varepsilon > 0$, the probability $p_{k,n}$ can be chosen in $[p - \varepsilon, p + \varepsilon]$ for suitable k and n, and that this choice can be made in polynomial time. The proof that the problem belongs to PSpace is easy, using standard techniques from the analysis of finite games. The idea is to generate the game tree in a depth-first manner, with a storage for paths (and some auxiliary information);

see [2,8]. In the remainder we treat only the hardness proof. The next section provides the indicated reduction, and in the subsequent section, we address the question of the distribution and efficient computation of the probabilities $p_{k,n}$.

3 PSPACE-Hardness of the Reachability Game

In order to prove the PSpace-hardness, we use a parametrized reduction from the problem Quantified Boolean Formulae (QBF), which is known to be PSpace-complete (cf. [8], problem "QSAT"). The reduction uses parts of the construction of Löding and Rohde [5]. The basic strategy is to construct an arena in such a way that, in a first part of the game, Runner can select the assignments for existential quantified variables of the formula, and that he is forced to choose certain assignments for the universal quantified variables. Then, this assignment is verified in a second part.

Formally, an instance of the problem QBF is a special quantified boolean formula, more precisely: Given a boolean expression ϑ in conjunctive normal form over boolean variables $x_1, \dots, x_m$, is $\exists x_1 \forall x_2 \dots Q_m x_m \vartheta$ true? Without loss of generality, one requires the formula to start with an existential quantifier. If m is even, $Q_m = \forall$; otherwise $Q_m = \exists$. For each instance φ of QBF, we construct a game arena $\mathcal{G}_{\varphi,k,n}$ and a rational number $p_{k,n}$ such that φ is true iff Runner has a strategy to reach a final vertex in $\mathcal{G}_{\varphi,k,n}$ exactly with probability $p_{k,n}$. Thereby the parameter k is an arbitrary natural number ≥ 2, and the parameter $n \in \mathbb{N}$ essentially has to be greater than the size of φ, i.e. $n \geq c \cdot |\varphi|$ for some constant c. If Runner plays suboptimally or if the formula is false, the maximum probability of winning is strictly lower than $p_{k,n}$; so the reduction meets the formulation of our game problem.

The arena consists of four types of subparts or "gadgets": The parametrization, existential, universal, and verification gadgets. In the parametrization gadget, one can, by adding or removing edges, adjust the probability $p_{k,n}$.

The outline of the proof is the following: We first introduce a construction to simulate a kind of multi-edge; this is convenient for a feasible analysis of the probabilistic outcome in a framework where only single edges are allowed. Then, we briefly recall the construction for the existential, universal, and verification gadgets [5], and adapt the argument to meet our model of a play against nature. In a second step, we introduce the parametrization gadget to adjust the probability $p_{k,n}$. Finally, we use this construction to prove our main result (Theorem 2.1).

3.1 The l-Edge Construction

In the following proofs, it will be necessary to link two vertices u and v in such a way that the connection is rather strong, i.e. there needs to be a number of several subsequent deletions until Runner is no longer able to move from u to v or vice versa. This is achieved by the construction shown in Figure 1, which we will call an "l-edge" from now on ($l \geq 1$). The circled vertex is a goal belonging to the

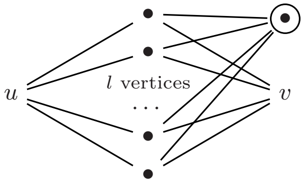


Fig. 1. An l-edge from u to v

set F of final vertices. Here, if after $|l|$ deletions all the $|l|$ edges between u and the middle vertices are deleted, Runner is disconnected from v. If $|l-1|$ or less edges have been deleted anywhere in the game graph, there is at least one path from u over some middle vertex to v and an edge between that middle vertex and the final vertex. Then, Runner can move from u (resp. v) to that middle vertex and is able to reach v (resp. u) in the next step, or he wins immediately by moving to the (circled) final vertex.

Lemma 3.1. *Given a randomized sabotage game with an l-edge between two nodes u and v, Runner can guarantee to reach v via this l-edge iff he can guarantee to reach u within $l-1$ moves.*

For a sufficiently high l, depending on the structure of the game-arena, it is clear that Runner cannot be hindered from winning by edge deletions in an l-edge; such l-edges will be called "∞-edges" to indicate that destruction of the connection can be considered impossible. (For classical sabotage games, l can be bounded by the total number of vertices in the game arena, because if Runner has a winning strategy in a classical sabotage game, he has also a winning strategy without visiting any vertex twice [5]. For the constructions in this paper where ∞-edges are used, it will be sufficient to consider ∞-edges as 4-edges.)

Remark 3.2. In order to sharpen the hardness result of this paper to randomized sabotage games with a *unique* goal, one may intend to merge final vertices to one vertex. But this is not always possible: Consider an l-edge to a final vertex v. Since we do not consider graphs with multi-edges, v cannot be merged with the other final vertex from the l-edge construction without breaking Lemma 3.1. For this reason, in this paper, PSpace-hardness is shown only for randomized sabotage games with at least two final vertices.

3.2 Existential, Universal, and Verification Gadgets

In this section, we briefly recall constructions from [5] to have a self-contained exposition. We introduce the existential and universal gadgets (applied according to the quantifier structure of the given formula), and the verification gadget (corresponding to its quantifier-free kernel).

The Existential Gadget: Intuitively, the existential component allows Runner to set an existential-quantified variable to true or false. The gadget is depicted

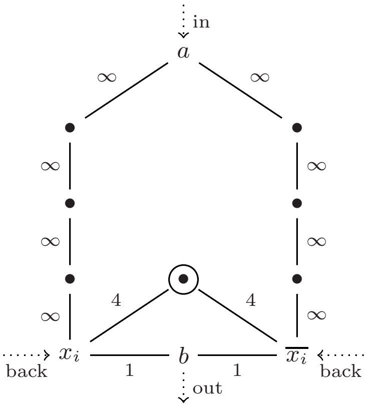


Fig. 2. The $\exists$-gadget for x_i with i odd

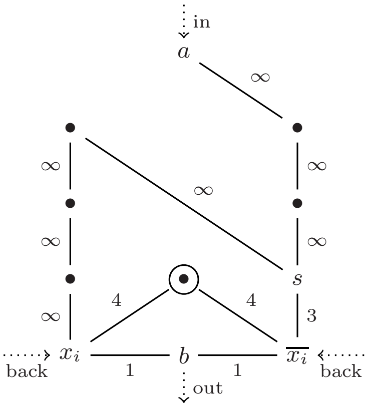


Fig. 3. The $\forall$-gadget for x_i with i even

in Figure 2. The node a is the input vertex for this gadget, and x_i (resp. $\overline{x_i}$) is the variable vertex of this component which Runner intends to visit if he sets x_i to false (resp. true). The vertex b is the exit node of this gadget; it coincides with the in-vertex of the next gadget (i.e. the universal gadget for x_{i+1}, or the verification gadget if x_i is the last quantified variable). The "back"-edges from x_i and $\overline{x_i}$ lead directly to the last gadget of the construction, the verification gadget. Later, Runner possibly move back via these edges to verify his assignment of the variable x_i. (We will see later that taking these edges as a shortcut, starting from the existential gadget, directly to the verification gadget, is useless for Runner.)

Of course, Runner has a very high probability of winning the game within the existential gadget (especially in an l-edge construction for an ∞-edge). But we are only interested in the worst-case scenario, where edges are deleted in the following precise manner:

When it is Runner's turn and he is currently residing in vertex a, he will move either left or right and can reach x_i (resp. $\overline{x_i}$) in four turns. When Runner moves towards x_i (resp. $\overline{x_i}$), the 4-edge from x_i (resp. $\overline{x_i}$) to the final vertex may be subsequently subject to deletion so that Runner ends up at node x_i (resp. $\overline{x_i}$) with no connection to the final vertex left. If Runner then moves towards $\overline{x_i}$ (resp. x_i) and the edge between b and $\overline{x_i}$ (resp. x_i) is deleted, he is forced to exit the gadget via b and move onwards. The 4-edge from $\overline{x_i}$ (resp. x_i) to the final vertex remains untouched so far. If Runner is later forced to move back to x_i or $\overline{x_i}$ from the verification gadget, he can only guarantee a win in one of these vertices.

The Universal Gadget: In the universal component a truth value for the all-quantified variables is chosen arbitrarily, but this choice can be considered to Runner's disadvantage in the worst-case. Runner can be forced to move in one or the other direction and has to set x_i to true or false, respectively. The gadget is depicted in Figure 3. A path through this gadget starts in node a and is intended to exit via node b, which coincides with the in-vertex of the next gadget (i.e. the existential gadget for x_{i+1}, or the verification gadget if x_i is the last quantified variable). Again, only the worst cases are important for now; Runner is able to win the game immediately in all other cases. Clearly, Runner is going to move in the direction of vertex s. There are two interesting scenarios which may happen with a probability > 0:

In the first scenario, the 3-edge to $\overline{x_i}$ is deleted completely. Then, Runner can only guarantee to leave the gadget at b via x_i (but no visit of $\overline{x_i}$ or the (circled) final vertex), because the 4-edge from x_i to the final vertex and the 1-edge to $\overline{x_i}$ may be deleted successively. In this case, the 4-edge between $\overline{x_i}$ and the final vertex remains untouched.

In the second case, only the 4-edge from $\overline{x_i}$ to the final vertex is subject to deletion. At s, Runner is intended to move downward to $\overline{x_i}$ and leave the gadget at b. Thereby the 4-edge between $\overline{x_i}$ and the final vertex (which was already reduced to a 1-edge) is deleted completely, and after this, the 1-edge to x_i is deleted. Consequently, the 4-edge from x_i to the final vertex is untouched. If Runner "misbehaves" in the sense that he moves from s to the left, it may happen that the final vertex becomes completely disconnected from both x_i and $\overline{x_i}$; in this case, Runner cannot win in this vertex if he is forced to move back to x_i or $\overline{x_i}$ from the verification gadget.

The Verification Gadget: The verification gadget is constructed in such a way that, when Runner arrives here, he can only force a win if the assignment for the variables which has been chosen beforehand satisfies the quantifier-free kernel of the formula.

The verification gadget for a QBF-formula with k clauses $C_1, \ldots, C_k$ is depicted in Figure 4. Its in-vertex coincides with exit vertex b of the last existential/universal gadget. For a clause $C_i = (\neg)x_{i_1} \vee (\neg)x_{i_2} \vee (\neg)x_{i_3}$, there are three paths, each from c_i via a single edge and an ∞-edge (the literal edge L_{ij}) back to the variable vertex x_{i_j} in the corresponding gadgets. Again, a look at the interesting scenarios is important:

Assume that Runner has chosen the appropriate assignments of the variables for a satisfiable formula. He reaches the first selection vertex s_1 via the ∞-edge

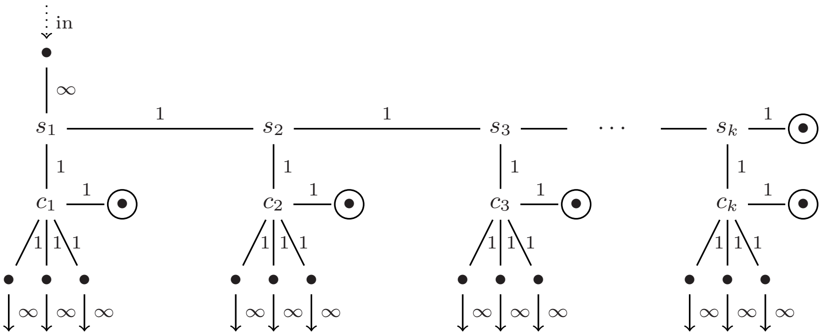


Fig. 4. The verification gadget for a formula with k clauses

from the last existential/universal gadget. If Runner is in s_i and the edge to c_i is deleted, he has to proceed to s_{i+1}. Now, assume that the edge between s_i and s_{i+1} is removed. Then, Runner is forced to move towards c_i. If the quantifier-free kernel of the QBF-formula is satisfied with the chosen assignment of Runner, then there is at least one literal that satisfies the clause C_i. Runner chooses to move alongside the corresponding literal edge L_{ij} back to x_{i_j}, into the gadget where he has chosen the assignment (and wins there by moving to the final vertex). In any other case Runner is able to win immediately by moving via s_k or via s_i, c_i to the (circled) final vertex. Note that he only has a chance of *always* winning if his chosen assignment actually fulfills the quantifier-free kernel of the formula.

If he did not choose a correct assignment or if the formula is not satisfiable, there is at least one clause that falsifies the QBF-formula, say clause c_i. If he is forced to go to c_i, there may be no path that leads him back to the final vertex of an existential/universal gadget.

As a side remark, one should note that it is never optimal for Runner to take a "back"-edge (i.e. a literal edge L_{ij}) as a shortcut, moving directly from some x_i (resp. $\overline{x_i}$) of an existential/universal gadget to the verification gadget. In this case, the 1-edge connecting c_i and the ∞-edge from x_i (resp. $\overline{x_i}$) to the verification gadget may be destroyed. Then, Runner has to move back and loses his chance of always winning the game.

We introduced so far the construction from [5] which suffices to show PSPACE-hardness for $p = 1$: Using the gadgets above, Runner does have a winning strategy iff the given formula is true. For a QBF-formula φ, we call the game arena of this construction $\mathcal{G}_\varphi$.

3.3 The Parametrization Gadget

The parametrization gadget is the initial part of the game arena that is constructed; it is used to "adjust" the overall winning probability of Runner. Runner starts from the initial vertex in this gadget. For each $k \geq 1$, we define the parametrization gadget $\mathcal{H}_k$; it is depicted in Figure 5.

We reduce the question of whether the QBF-formula φ is true to the reachability problem over certain game arenas that result from combining $\mathcal{H}_k$ with $\mathcal{G}_\varphi$

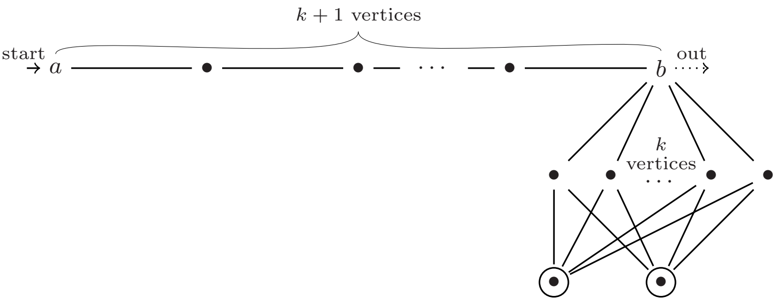


Fig. 5. The parametrization gadget $\mathcal{H}_k$

as a graph $\mathcal{H}_k \circ \mathcal{G}_\varphi$: The out-vertex b in $\mathcal{H}_k$ is identified with the in-vertex a of the first existential gadget of $\mathcal{G}_\varphi$. Assume $\mathcal{G}_\varphi$ has n_0 edges. We get modifications of $\mathcal{G}_\varphi$ with any number $n \geq n_0$ of edges by adding artificial extra edges (without changing the behavior in the discussed worst case); for instance, this can be achieved by adding a path with a dead end. We call this game arena $\mathcal{G}_\varphi^n$. In the sequel, we work with

$$\mathcal{G}_{\varphi,k,n} := \mathcal{H}_k \circ \mathcal{G}_\varphi^n \ .$$

Since $\mathcal{H}_k$ has $4k$ edges, the overall number of edges in our game arena $\mathcal{G}_{\varphi,k,n}$ is $4k + n$. Let

$$\begin{aligned} p_{k,n} := &\text{ probability for Runner to traverse the} \\ &\text{ parametrization gadget } \mathcal{H}_k \text{ of } \mathcal{G}_{\varphi,k,n} \ . \end{aligned}$$

Lemma 3.3. *Runner wins the randomized reachability game over $\mathcal{G}_{\varphi,k,n}$ for probability $p_{k,n}$ iff the QBF-formula φ is true.*

Proof. First note the following: If Runner resides at vertex b, and in $\mathcal{H}_k$ exactly the k edges below vertex b have been deleted so far, then Runner wins with probability 1 iff φ is true (this follows immediately from the classical construction by Löding and Rohde [5]).

Now assume that φ is true. In the parametrization gadget Runner starts at node a and obviously moves towards b. Clearly, if any edge between his current position and b is deleted, he loses the game immediately. However, if he succeeds in getting to b, he will always win the game: First, assume the case that, in his k steps towards b, only edges in $\mathcal{H}_k$ were subject to deletion. Then, Runner always wins by moving in the first existential gadget and traversing $\mathcal{G}_\varphi^n$, as mentioned before. If we assume the other case that at least one of the k deletions took place outside of $\mathcal{H}_k$, there is at least one edge leading from b downward to some middle node, say b', and there are at least two edges leading from b' to the two final vertices in the parametrization gadget. Then, Runner wins by moving from b downward to b' and then, depending on the next deletion, by moving to one of the two final vertices. In both cases, Runner wins over $\mathcal{G}_{\varphi,k,n}$ exactly with the probability $p_{k,n}$ of traversing the parametrization gadget $\mathcal{H}_k$ from node a to node b.

Now, assume that φ is false, and that only the k edges below vertex b in $\mathcal{H}_k$ are subject to deletion while Runner moves towards b (this may happen with a probability > 0). Then, Runner's only chance to win from b is by moving towards some final vertex in $\mathcal{G}_\varphi^n$. Since φ is false, his winning probability in b is strictly smaller than 1, and hence his overall winning probability for $\mathcal{G}_{\varphi,k,n}$ is strictly smaller than $p_{k,n}$. □

The computation of the probabilities $p_{k,n}$ only depends on the parametrization gadget $\mathcal{H}_k$ and n: Clearly $p_{1,n} = 1$. For $k \geq 2$, the winning probability is obtained from the probability of not failing in the first step, multiplied by the probability of not failing in the second step, etc., until the probability of not failing in the $(k-2)$-th step, where Runner tries to get to b within one step. After the first step, Runner has still to cross $k-1$ edges; neither of them may be deleted. Overall, there are $4k+n$ edges, so Runner does not lose if any of the other $4k+n-(k-1)$ edges is deleted. In the last step before reaching b, $k-2$ edges have been deleted, so $4k+n-(k-2)$ edges are left in the game. If any other than the edge between Runner's current position and b is deleted, Runner is able to reach b. Generally, in the i-th step there are $4k+n-(i-1)$ edges left; and in order for Runner to still be able to reach b, one of the $3k+n+1$ edges that are not between Runner's current position and b has to be deleted. Altogether,

$$p_{k,n} = \frac{3k+n+1}{4k+n} \cdots \frac{3k+n+1}{4k+n-(k-2)} = \prod_{i=0}^{k-2} \frac{3k+n+1}{4k+n-i} \ .$$

We can summarize these observations in the following theorem:

Theorem 3.4. *Given a QBF-formula φ so that $\mathcal{G}_\varphi$ has n_0 edges, for all $k,n \in \mathbb{N}$ with $k \geq 2$ and $n \geq n_0$ the following holds: Runner wins the randomized reachability game over $\mathcal{G}_{\varphi,k,n}$ for probability $p_{k,n} = \prod_{i=0}^{k-2} \frac{3k+n+1}{4k+n-i}$ iff the QBF-formula φ is true.*

In order to use this theorem for a reduction to the randomized sabotage game, we need to show that the game arena can be constructed in polynomial time. In the following Lemma, we show that the size of the constructed game $\mathcal{G}_{\varphi,k,n}$ is linear in the size of the inputs φ, k, and n:

Lemma 3.5. *The size of $\mathcal{G}_\varphi$ is linear in $|\varphi|$, and the size of $\mathcal{H}_k$ is linear in k.*

Proof. For the first part, it is sufficient to realize that the size of each gadget can be bounded by a constant. Since the number of gadgets is linear in the size of φ, the number of vertices and edges of $\mathcal{G}_\varphi$ is linear in $|\varphi|$. The only problem might be the ∞-edges; but by detailed observation, we see that each ∞-edge can be replaced by a 4-edge, and the construction still works in the same way.

For the second part, it suffices to note that $\mathcal{H}_k$ has $2k+3$ vertices and $4k$ edges. □

Now, a preliminary result can be formulated in the following form:

Corollary 3.6. *For all $k \geq 2$, the following problem is* PSpace*-hard: Given a randomized sabotage game with n edges, does Runner win with a probability $\geq p_{k,n}$?*

3.4 Towards the PSPACE-Hardness for Arbitrary Probabilities

We already have a reduction of QBF to randomized sabotage games with a varying probability $p_{k,n}$ (which depends on the given game graph). By a closer look at the term $p_{k,n}$ we see that the probability $p_{k,n}$ can be adjusted arbitrary close to 0 and arbitrary close to 1; more precisely: For a fixed k, we have $\lim_{n\to\infty} p_{k,n} = 1$; and for a fixed n, we have $\lim_{k\to\infty} p_{k,n} = 0$. We will show a stronger result, namely that the probabilities $p_{k,n}$ form a dense set in the interval $[0,1]$, and that k and n can be computed efficiently such that $p_{k,n}$ is in a given interval. More precisely, we shall show the following:

Theorem 3.7. *The set of probabilities* $\{p_{k,n} \mid k, n \in \mathbb{N},\ k \geq 2\}$ *is dense in the interval* $[0, 1]$. *Moreover, given* $n_0 \in \mathbb{N}$, $p \in [0, 1]$, *and an* $\varepsilon > 0$, *there exist* $k, n \in \mathbb{N}$ *with* $k \geq 2$ *and* $n \geq n_0$ *such that* $p_{k,n} \in [p - \varepsilon, p + \varepsilon]$; *the computation of such* k, n, *and* $p_{k,n}$ *is polynomial in the* numerical values *of* n_0, $\frac{1}{p}$, *and* $\frac{1}{\varepsilon}$.

The proof of this theorem is the subject of Section 4.

Note that Theorem 3.7 provides a pseudo-polynomial algorithm, since the computation is only polynomial in the *numerical values* of n_0, $\frac{1}{p}$, and $\frac{1}{\varepsilon}$ (and not in their *lengths*, which are logarithmic in the numerical values). For our needs – i.e. a polynomial time reduction to prove Theorem 2.1 – this is no restriction: The parameter n_0 corresponds to the number of edges in the input game (which has already a polynomial representation), and p and ε are fixed values (i.e. formally they do not belong to the problem instance).

Now, we prove our main result:

Proof (of Theorem 2.1). For arbitrary p and ε, we give a reduction from QBF to the randomized sabotage game problem where only probabilities in the interval $[p-\varepsilon, p+\varepsilon]$ are allowed. Note that p and ε do not belong to the problem instance; so they are considered constant in the following.

Given a QBF-formula φ, we need to compute a game $\mathcal{G}_{\varphi,k,n}$ and a $p_{k,n} \in [p-\varepsilon, p+\varepsilon]$ such that Runner wins $\mathcal{G}_{\varphi,k,n}$ with a probability $\geq p_{k,n}$ iff φ is true. Given φ, we first apply the construction of Section 3.2 to construct an equivalent sabotage game $\mathcal{G}_\varphi$. Let n_0 be the number of edges of $\mathcal{G}_\varphi$, which is linear in the size of φ according to Lemma 3.5. Then, we can compute $k \geq 2$, $n \geq n_0$, and $p_{k,n}$ according to Theorem 3.7. For a fixed ε, the computations are polynomial in n_0 and hence polynomial in $|\varphi|$. Now, we extend $\mathcal{G}_\varphi$ to an equivalent sabotage game with n edges, denoted $\mathcal{G}^n_\varphi$. This can be achieved by adding $n - n_0$ dummy-edges (e.g. we can add a path with a dead end). Thereafter, we construct the randomized sabotage game $\mathcal{G}_{\varphi,k,n}$ by combining the parametrization gadget $\mathcal{H}_k$ with the game arena $\mathcal{G}^n_\varphi$.

The claimed equivalence of φ to the stated randomized reachability game problem for probability $p_{k,n}$ holds due to Theorem 3.4. The requirement that $p_{k,n}$ is in the interval $[p - \varepsilon, p + \varepsilon]$ follows from Theorem 3.7. □

4 On the Distribution and Computation of the Probabilities $p_{k,n}$

This section deals with the proof of Theorem 3.7: Given $n_0 \in \mathbb{N}$, $p \in [0, 1]$, and an $\varepsilon > 0$, we can construct $k > 2$ and $n \geq n_0$ in polynomial time with respect to the numerical values of n_0, $\frac{1}{p}$, and $\frac{1}{\varepsilon}$, such that $p_{k,n}$ is in the interval $[p-\varepsilon, p+\varepsilon]$.

The idea is to first adjust the probability $p_{k,n}$ arbitrary close to 1, and then go with steps of length below any given $\varepsilon > 0$ arbitrary close to 0; so, we hit every ε-neighborhood in the interval $[0, 1]$.

In order to adjust the probability $p_{k,n}$ arbitrary close to 1, we first choose $k = 2$ and a sufficiently high $n \geq n_0$. (We can artificially increase n by adding a path with a dead end.) We will show that it suffices to choose $n := \max\{n_0, \lceil\frac{1}{\varepsilon}\rceil\}$. For this choice we obtain $p_{2,n} \geq 1-\varepsilon$ ($\geq p-\varepsilon$). Then, we decrease the probability by stepwise incrementing k by 1 (changing $\mathcal{H}_k$ to $\mathcal{H}_{k+1}$ and keeping n constant). It will turn out that (with the choice of n as above) the probability decreases by a value that is lower than $\frac{1}{4k+n+4}$ ($\leq \varepsilon$). Iterating this, the values converge to 0, and we hit the interval $[p-\varepsilon, p+\varepsilon]$. Hence, the set of probabilities $\{p_{k,n} \mid k,n \in \mathbb{N},\ k \geq 2\}$ is dense in the interval $[0,1]$. Furthermore, we will show that it will be sufficient to increase k at most up to $8n$. For this choice, we obtain $p_{k,n} \leq \varepsilon$ ($\leq p+\varepsilon$).

For the complexity analysis, note the following: After each step, the algorithm has to check efficiently whether $p_{k,n} \in [p-\varepsilon, p+\varepsilon]$. The computation of the term $p_{k,n}$ is pseudo-polynomial in k, n, and the test for $p_{k,n} \leq p+\varepsilon$ is in addition polynomial in $\frac{1}{p}$ and $\frac{1}{\varepsilon}$. Since k and n are pseudo-linear in n_0 and $\frac{1}{\varepsilon}$, the whole procedure is pseudo-polynomial in n_0, $\frac{1}{p}$, and $\frac{1}{\varepsilon}$.

Four claims remain to be proved:

- The adjustment of $p_{k,n}$ arbitrary close to 1 with the proposed choice of n, i.e. given $\varepsilon > 0$, for $n \geq \frac{1}{\varepsilon}$ holds $p_{2,n} \geq 1-\varepsilon$.
- The adjustment of $p_{k,n}$ arbitrary close to 0 with the proposed choice of k, i.e. given $\varepsilon > 0$ and $n \geq \frac{1}{\varepsilon}$, for $k \geq 8n$ holds $p_{k,n} \leq \varepsilon$.
- The estimation $p_{k,n} - p_{k+1,n} < \frac{1}{4k+n+4}$.
- The test for $p_{k,n} \in [p-\varepsilon, p+\varepsilon]$ is pseudo-polynomial in k, n, $\frac{1}{p}$ and $\frac{1}{\varepsilon}$.

These claims are shown in the rest of this section:

Lemma 4.1. *Given $\varepsilon > 0$, for $n \geq \lceil\frac{1}{\varepsilon}\rceil$ we have $p_{2,n} \geq 1-\varepsilon$.*

Proof. Since $n \geq \lceil\frac{1}{\varepsilon}\rceil \geq \frac{1}{\varepsilon} \geq \frac{1}{\varepsilon} - 8$ for $\varepsilon > 0$, the result follows from

$$p_{2,n} = \frac{n+7}{n+8} \geq 1-\varepsilon \iff n \geq \frac{1}{\varepsilon} - 8\ .$$

□

Lemma 4.2. *Given $\varepsilon > 0$ and $n \in \mathbb{N}$ with $n \geq \frac{1}{\varepsilon}$ and $n \geq 4$, for $k \geq 8n$ we have $p_{k,n} < \varepsilon$.*

Proof. First note that we have at least $n \geq 1$ and $k \geq 8$. Then

$$\begin{aligned} p_{k,n} &= \prod_{i=0}^{k-2} \frac{3k+n+1}{4k+n-i} \leq \left(\frac{3k+n+1}{3.5k+n}\right)^{\frac{k}{2}} \leq \left(\frac{4k+1}{4.5k}\right)^{\frac{k}{2}} \leq \left(\frac{4.125k}{4.5k}\right)^{\frac{k}{2}} \\ &= \left(\frac{11k}{12k}\right)^{\frac{k}{2}} = \left(\frac{11}{12}\right)^{\frac{k}{2}} \leq \left(\frac{11}{12}\right)^{4n} < \varepsilon\ . \end{aligned}$$

The inequality $\left(\frac{11}{12}\right)^{4n} < \varepsilon$ remains to be shown. Since $\frac{1}{n} \leq \varepsilon$, it is sufficient to show that $\left(\frac{11}{12}\right)^{4n} < \frac{1}{n}$:

$$\left(\frac{11}{12}\right)^{4n} < \frac{1}{n} \iff n^{\frac{1}{4n}} < \frac{12}{11} \iff \sqrt[n]{n}^{\frac{1}{4}} \leq \sqrt{2}^{\frac{1}{4}} < \frac{12}{11}\ .$$

The inequality $\sqrt[n]{n}^{\frac{1}{4}} \leq \sqrt{2}^{\frac{1}{4}}$ is equivalent to $n^2 \leq 2^n$ and holds for all $n \geq 4$. □

Lemma 4.3. *For $k, n \in \mathbb{N}$ with $k \geq 2$, we have $p_{k,n} - p_{k+1,n} < \frac{1}{4k+n+4}$.*

Proof. In this proof we use the substitution $m := 4k + n + 4$.

$$p_{k,n} - p_{k+1,n} = \prod_{i=0}^{k-2} \frac{3k+n+1}{4k+n-i} - \prod_{i=0}^{k-1} \frac{3k+n+4}{4k+n+4-i}$$

$$\leq \prod_{i=0}^{k-2} \frac{3k+n+4+1}{4k+n+4-i} - \prod_{i=0}^{k-1} \frac{3k+n+4}{4k+n+4-i} = \prod_{i=0}^{k-2} \frac{m-k+1}{m-i} - \prod_{i=0}^{k-1} \frac{m-k}{m-i}$$

$$= \prod_{i=0}^{k-1} \frac{m-k+1}{m-i} - \prod_{i=0}^{k-1} \frac{m-k}{m-i} = \frac{(m-k+1)^{k-1} - (m-k)^{k-1}}{\prod_{i=0}^{k-1} m-i} \ .$$

Now we can use the equation $a^l - b^l = (a-b)(a^{l-1} + a^{l-2}b + \cdots + ab^{l-2} + b^{l-1})$ for the estimation $(d+1)^{k-1} - d^{k-1} = (d+1)^{k-2} + (d+1)^{k-3}d + \cdots + (d+1)d^{k-3} + d^{k-2}$ $\leq (k-1)(d+1)^{k-2}$. We obtain $p_{k,n} - p_{k+1,n}$

$$\leq \frac{(k-1)(m-k+1)^{k-2}}{\prod_{i=0}^{k-1} m-i} = \frac{k-1}{m(m-1)} \prod_{i=2}^{k-1} \frac{(m-k+1)}{m-i} \leq \frac{k-1}{m(m-1)} \ .$$

Since $m > k$ for all $k, n \in \mathbb{N}$, we obtain $p_{k,n} - p_{k+1,n} < \frac{1}{m} = \frac{1}{4k+n+4}$. □

Lemma 4.4. *The computation of the term $p_{k,n}$ is pseudo-polynomial in k and n. The test for $p_{k,n} \leq p + \varepsilon$ is pseudo-polynomial in k and n, and polynomial in $\frac{1}{p}$ and $\frac{1}{\varepsilon}$.*

Proof. First, we rewrite $p_{k,n}$ in the form

$$\frac{(3k+n+1)^{k-1}}{\prod_{i=0}^{k-2} 4k+n-i} \ .$$

Now, we compute the numerator and the denominator separately. For the computation, we can switch to binary encoding. Each multiplication can be performed in polynomial time in the length of its binary encoding [13]. We need $k-2$ multiplications (for this reason, the algorithm is only pseudo-polynomial). The division and comparison of two rational numbers can be done in polynomial time with respect to the length of their binary representations [13]. So, the quotient $p_{k,n}$ can be computed in pseudo-polynomial time with respect to k and n, and the test to check whether $p_{k,n} \leq p + \varepsilon$ is in addition polynomial in $\frac{1}{p}$ and $\frac{1}{\varepsilon}$. □

5 Perspectives

We have introduced randomized sabotage games, and showed that the reachability problem for a probability which may vary in a fixed interval $[p - \varepsilon, p + \varepsilon]$ is PSpace-complete. This is a small contribution to the emerging research on the analysis of dynamical networks with aspects of randomness. As concrete open issues, we mention the following problems:

1. In our proof, it seems difficult to adjust the probability *exactly* to a given probability p (in our formulation this is the case $\varepsilon = 0$). It remains open whether this can be achieved by a refinement of the construction.
2. In our proof, we used the reachability problem with a target set F containing at least two vertices (see Remark 3.2). One task is to extend the result to cover also the case of a singleton as target set (note that in the non-randomized case, the singleton reachability problem is PSpace-hard only if one allows multi-edges [5]).
3. The proof of our model depends on the restriction that exactly one edge per turn is deleted (rather than possibly multiple edge deletion occurring subject to given probabilities). Sharpening the mentioned problem of "dynamic graph reliability" [7] to probabilities that are independent of Runner's position, we can study the model where in every turn each edge fails with a probability $p(e)$, or even with probability $\frac{1}{n}$ in the uniform case.
4. Extending the model with a mechanism of restoration is a challenging task (for instance, *reactive Kripke models* [1] and *backup parity games* [11] address this issue). In [9] we developed a theory of dynamic networks where Runner and Blocker are replaced by two players, *Constructor* and *Destructor*, that add resp. delete vertices/edges, and the problem of guaranteeing certain network properties (like connectivity) is addressed. We are presently integrating probabilistic features into this model, starting from the present paper. Another interesting direction of research is to include more general winning conditions in appropriate logics [2].

Acknowledgments. We thank Łukasz Kaiser for his help regarding Lemma 4.3. We also thank the anonymous referees for their valuable comments and suggestions in improving this paper.

References

1. Gabbay, D.M.: Introducing reactive kripke semantics and arc accessibility. In: Avron, A., Dershowitz, N., Rabinovich, A. (eds.) Pillars of Computer Science. LNCS, vol. 4800, pp. 292–341. Springer, Heidelberg (2008)
2. Klein, D.: Solving Randomized Sabotage Games for Navigation in Networks. Diploma thesis, RWTH Aachen (2008)
3. Littman, M.L., Majercik, S.M., Pitassi, T.: Stochastic boolean satisfiability. Journal of Automated Reasoning 27(3), 251–296 (2001)
4. Löding, C., Rohde, P.: Model checking and satisfiability for sabotage modal logic. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 302–313. Springer, Heidelberg (2003)
5. Löding, C., Rohde, P.: Solving the sabotage game is PSPACE-hard. Technical Report AIB-05-2003, RWTH Aachen (2003)

6. Löding, C., Rohde, P.: Solving the sabotage game is PSPACE-hard. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 531–540. Springer, Heidelberg (2003)
7. Papadimitriou, C.H.: Games against nature. Journal of Computer and System Sciences 31(2), 288–301 (1985)
8. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley, Reading (1994)
9. Radmacher, F.G., Thomas, W.: A game theoretic approach to the analysis of dynamic networks. In: Poetzsch-Heffter, A., Schneider, K. (eds.) Proceedings of VerAS. Electronic Notes in Theoretical Computer Science, vol. 200(2), pp. 21–37. Elsevier, Amsterdam (2008)
10. Rohde, P.: Moving in a crumbling network: The balanced case. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 310–324. Springer, Heidelberg (2004)
11. Rohde, P.: On Games and Logics over Dynamically Changing Structures. PhD thesis, RWTH Aachen (2005)
12. van Benthem, J.: An essay on sabotage and obstruction. In: Hutter, D., Stephan, W. (eds.) Mechanizing Mathematical Reasoning. LNCS (LNAI), vol. 2605, pp. 268–276. Springer, Heidelberg (2005)
13. von zur Gathen, J., Gerhard, J.: Modern Computer Algebra, 2nd edn. Cambridge University Press, Cambridge (2003)

Applying Step Coverability Trees to Communicating Component-Based Systems

Jetty Kleijn[1] and Maciej Koutny[2]

[1] LIACS, Leiden University
P.O.Box 9512, NL-2300 RA Leiden, The Netherlands
kleijn@liacs.nl
[2] School of Computing Science, Newcastle University
Newcastle upon Tyne NE1 7RU, U.K.
maciej.koutny@ncl.ac.uk

Abstract. Like reachability, coverability is an important tool for verifying behavioural properties of dynamic systems. When a system is modelled as a Petri net, the classical Karp-Miller coverability tree construction can be used to decide questions related to the (required) capacity of local states. Correctness (termination) of the construction is based on a monotonicity property: more resources available implies more behaviour possible. Here we discuss a modification of the coverability tree construction allowing one to deal with concurrent occurrences of actions (steps) and to extend the notion of coverability to a dynamic action-based notion (thus viewing bandwidth as a resource). We are in particular interested in component-based systems in which steps are subject to additional constraints like (local) synchronicity or maximal concurrency. In general the behaviour of such systems is not monotonous and hence new termination criteria (depending on the step semantics) are needed. We here investigate marked graphs, a Petri net model for systems consisting of concurrent components communicating via buffers.

Keywords: Petri nets; step semantics; step coverability tree; boundedness; decidability; maximal concurrency; marked graphs; components; localities.

1 Introduction

Coverability can be applied as an important tool for verifying behavioural properties of dynamic systems with quantified state information — typically capturing the presence of certain kinds of resources — modelled as parallel program schemata, like Vector Addition Systems [12], Petri nets [17], or state machines communicating via buffers [5]. In this paper, Place/Transition Petri nets (PT-nets) are used as our basic system model. The dynamics of a PT-net derives from a 'firing rule' describing enabledness of individual actions i.e., the potential to occur at a global state or 'marking', and the effect such occurrence has on a marking. This sequential firing rule can then be extended to step firing rules for sets or multisets of simultaneously occurring transitions. Firing rules lead

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 178–193, 2010.

to behavioural descriptions of a PT-net in terms of firing or step sequences as well as reachability graphs (labelled transition systems) in which such execution sequences are combined with state information. The latter have proved to be very useful as they allow behavioural analysis and verification (including model checking [22]). An important property for verification purposes is the 'boundedness' of a PT-net which amounts to saying that its state space is finite. The standard construction to investigate boundedness is the 'coverability tree' (CT) introduced in [12] and investigated in e.g., [7,10,17,23]. CTs can be used to answer also questions related to boundedness of local states (resources) such as 'will there be enough resources available?' (e.g., to avoid deadlocks) or 'is the amount of resources generated unbounded?' (and hence restricting the capacity of certain parts of the system may constrain its behaviour). Similarly to reachability graphs or trees, CTs can be a tool for deciding other relevant behavioural questions as well, even in the case of infinite state spaces. The reason is that the constructed CT is always finite, with the termination of the construction being based on a 'monotonicity' property implying that no current behaviour is lost when more resources become available.

The standard CT is defined only for the interleaving (sequential) semantics of PT-nets and, as a consequence, issues relating to the step based semantics are not accurately reflected. To capture this aspect of concurrency, the concept of a 'covering' step or 'extended' step was introduced in [13]. The resulting step coverability tree (SCT) of a PT-net extends the behavioural information conveyed by the sequential CT, by providing a more concurrency-oriented view of the behaviour of the PT-net. Whereas the standard approach is concerned with the use of resources, here bandwidth is also a resource (steps may require unbounded capacity), and one may be interested in e.g., whether restricting the bandwidth of steps can lead to a restricted (or even incorrect) behaviour. Moreover, SCTs can be applied to other Petri net models or PT-nets operating under a step semantics involving more concurrency constraints. The sequential and standard step based semantics of PT-nets are in many respects equivalent. By the substep property each step can be sequentialised to a sequence of transitions with the same effect and so the reachable markings are the same for both semantics. However, there are practically relevant extensions of PT-nets for which this does not hold, such as PT-nets with inhibitor arcs and the a priori step semantics for which the construction of an SCT needs to be adjusted. However, for such nets even the standard CT construction no longer works, mainly because the presence of additional resources may constrain behaviour. Thus in [3], a CT construction has been developed for (a subclass of) PT-nets with inhibitor arcs. and in [13] this construction was extended to deal with the a priori step semantics.

In our research we are interested in coverability in the context of distributed systems consisting of communicating components. Systems of this kind often behave in a 'globally asynchronous locally synchronous' (GALS) manner implying that at the local level their computational progress is captured by a maximally concurrent step semantics. For such systems, an accurate behavioural representation can be provided by PT-nets with localities (PTL-nets [14]) with explicitly located transitions. A maximally concurrent semantics however does not in general satisfy the substep property nor will it be monotonous. Still, it can be viewed as monotonic in the weaker sense that adding resources can enable larger steps without invalidating already enabled transitions. This weak monotonicity can then perhaps be used to construct SCT for such systems. Here we undertake a first case study of this execution model, by looking at marked graphs, a basic Petri net model for systems with components communicating through buffers.

The paper is a follow-up to our investigations in [13] where we introduced SCTs more or less ad hoc for a class of inhibitor nets. It initiates a systematic investigation of SCTs, focussing on the correctness of the construction for systems where transitions occurring in steps are subject to additional synchronisation constraints based on the localities (of the components) they belong to. Here we explore the construction of step coverability trees for two extreme options: no synchrony i.e., the standard PT-net step semantics, and full synchrony i.e., maximal concurrency, for a simple class of nets without local choices. Hence we recall the construction of SCTs derived from [13], specialising the basic results and proofs to the case of PT-nets and thus making them more accessible and so amenable to possible modification. Next, we demonstrate how to construct SCTs for the class of marked graphs operating under the maximally concurrent semantics.

We use standard mathematical notation, in particular, $\uplus$ denotes disjoint set union, $\mathbb{N} = \{0, 1, 2, \ldots\}$ the set of natural numbers, and ω the first infinite ordinal. We assume that $\omega + \alpha = \omega - \alpha = k \cdot \omega = \omega$, $n - \omega = 0 \cdot \omega = 0$ and $n < \omega$, for $n \geq 0$, $k > 0$ and $\alpha \in \mathbb{N} \cup \{\omega\}$.

A *multiset* over a set X (in this paper always finite) is a function $\mu : X \to \mathbb{N}$, and an *extended multiset* (over X) is a function $\mu : X \to \mathbb{N} \cup \{\omega\}$. Any subset of X may be viewed through its characteristic function as a multiset over X, and a multiset may always be considered as an extended multiset. For an extended multiset μ, we write $x \in \mu$ if $\mu(x) > 0$. For a multiset μ over X, the cardinality of μ is defined as $|\mu| \overset{\text{df}}{=} \sum_{x \in X} \mu(x)$. We will use formal sums to denote extended multisets; thus we write $2{\times}a{+}b{+}\omega{\times}c$ for the multiset μ with $\mu(a) = 2$, $\mu(b) = 1$, $\mu(c) = \omega$, and $\mu(x) = 0$, for $x \neq a, b, c$. For an extended multiset μ over X, its ω-domain is $dom_\omega(\mu) = \{x \in X \mid \mu(x) = \omega\}$. Let μ and μ' be extended multisets over X. We write $\mu \leq \mu'$ and say that μ' *covers* μ if $\mu(x) \leq \mu'(x)$ for all $x \in X$. If $\mu(x) \leq \mu'(x)$ and $\mu(x) \neq \mu'(x)$, we write $\mu(x) < \mu'(x)$. Moreover, $(\mu + \mu')(x) \overset{\text{df}}{=} \mu(x) + \mu'(x)$, and $(\mu - \mu')(x) \overset{\text{df}}{=} \max\{0, \mu(x) - \mu'(x)\}$. The multiplication of μ by a natural number is given by $(n \cdot \mu)(x) \overset{\text{df}}{=} n \cdot \mu(x)$. If μ is a multiset over X, μ' an extended multiset over the same set X and $k \geq 0$, then we say that μ is a *k-approximation* of μ' if, for all $x \in X$, $\mu(x) = \mu'(x)$ if $\mu'(x) < \omega$, and otherwise $\mu(x) > k$. We denote this by $\mu \Subset_k \mu'$.

In proofs we may use implicitly *Dickson's Lemma* which states that every infinite sequence of extended multisets over a common finite set contains an infinite non-decreasing subsequence, and *König's Lemma* which states that every infinite, finitely branching tree has an infinite path starting from the root.

2 PT-Nets

A *net* is a triple $\mathcal{N} = (P, T, W)$ such that P and T are disjoint finite sets of *places* and *transitions*, respectively, and $W : (T \times P) \cup (P \times T) \to \mathbb{N}$ is the *weight function* of $\mathcal{N}$. In diagrams, places are drawn as circles and transitions as rectangles. If $W(x, y) \geq 1$ for some $(x, y) \in (T \times P) \cup (P \times T)$, then (x, y) is an *arc* leading from x to y. As usual, arcs are annotated with their weight if this is 2 or more. A double headed arrow between p and t indicates that $W(p, t) = W(t, p) = 1$. We assume that, for every $t \in T$, there is a place p such that $W(p, t) \geq 1$ or $W(t, p) \geq 1$, i.e., transitions are never isolated. (Requiring that transitions are not isolated instead of imposing the stronger condition that each transition has both at least one input place and at least one output place has no technical consequences, but it allows for smaller examples.)

Given a transition t of a net $\mathcal{N} = (P, T, W)$, we denote by $t^\bullet$ the multiset of places given by $t^\bullet(p) \overset{\text{df}}{=} W(t, p)$ and by ${}^\bullet t$ the multiset of places given by ${}^\bullet t(p) \overset{\text{df}}{=} W(p, t)$. Both notations extend to multisets U of transitions in the following way: $U^\bullet \overset{\text{df}}{=} \sum_{t \in U} U(t) \cdot t^\bullet$ and ${}^\bullet U \overset{\text{df}}{=} \sum_{t \in U} U(t) \cdot {}^\bullet t$. For a place p, we denote by ${}^\bullet p$ and $p^\bullet$ the multisets of transitions given by $p^\bullet(t) \overset{\text{df}}{=} W(p, t)$ and ${}^\bullet p(t) \overset{\text{df}}{=} W(t, p)$, respectively.

The states of a net $\mathcal{N} = (P, T, W)$ are given as multisets of places, so-called *markings*. Given a marking M of $\mathcal{N}$ and a place $p \in P$, we say that p is marked (under M) if $M(p) \geq 1$ and that $M(p)$ is the number of tokens in p. In diagrams, every token in a place is drawn as a small black dot. Also, if the set of places of $\mathcal{N}$ is implicitly ordered, $P = \{1, \ldots, n\}$, then we will represent any marking M of $\mathcal{N}$ as the n-tuple $(M(1), \ldots, M(n))$ of natural numbers.

Transitions represent actions which may occur at a given marking and then lead to a new marking. First, we discuss the *sequential semantics* of nets.

A transition t of $\mathcal{N} = (P, T, W)$ can occur at a marking M of $\mathcal{N}$ if for each place p, the number of tokens $M(p)$ in p is at least $W(p, t)$. Formally, t is *enabled* at M, denoted by $M[t\rangle$, if ${}^\bullet t \leq M$. If t is enabled at M, then it can be *executed* (or *fired*) leading to the marking $M' \overset{\text{df}}{=} M - {}^\bullet t + t^\bullet$, denoted by $M[t\rangle M'$. Thus M' is obtained from M by deleting $W(p, t)$ tokens from each place p and adding $W(t, p)$ tokens to each place p.
A *firing sequence* from a marking M to marking M' in $\mathcal{N}$ is a possibly empty sequence of transitions $\sigma = t_1 \ldots t_n$ such that $M = M_0 \, [t_1\rangle \, M_1 \, [t_2\rangle M_2 \cdots M_{n-1} \, [t_n\rangle \, M_n = M'$, for some markings $M_1, \ldots, M_{n-1}$ of $\mathcal{N}$. Note, that if σ is the empty firing sequence, then $M = M'$. If σ is a firing sequence from M to M', then we write $M \, [\sigma\rangle_{\mathit{fs}} \, M'$ and call M' *fs–reachable* from M (in $\mathcal{N}$).

Figure 1(a) shows a net with marking $(1, 0, 0)$. It has infinitely many non-empty firing sequences starting from $(1, 0, 0)$, such as $\sigma_1 = t$, $\sigma_2 = u$, $\sigma_3 = uv$, and $\sigma_4 = uvv$. The set of markings *fs*–reachable from the marking $(1, 0, 0)$ is also infinite and it comprises, for example, $(1, 1, 1)$, $(0, 1, 0)$, $(0, 1, 1)$, and $(0, 1, 2)$.

Next we define a semantics of nets in terms of concurrently occurring transitions. A *step* of a net $\mathcal{N} = (P, T, W)$ is a multiset of transitions, $U : T \to \mathbb{N}$.

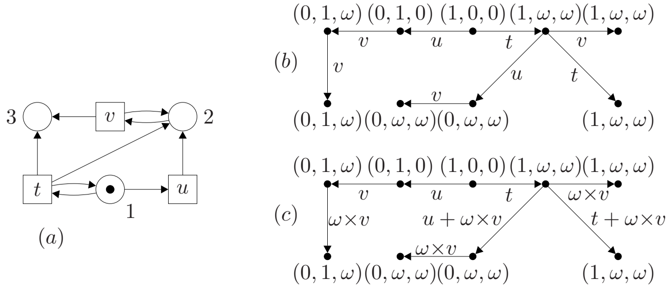

3
v
2
t
1
u
(a)
(0, 1, ω) (0, 1, 0) (1, 0, 0) (1, ω, ω) (1, ω, ω)
v
u
t
v
(b)
v
u
t
v
(0, 1, ω) (0, ω, ω) (0, ω, ω)
(1, ω, ω)
(0, 1, ω) (0, 1, 0) (1, 0, 0) (1, ω, ω) (1, ω, ω)
v
u
t
ω×v
(c)
ω×v
u + ω×v
t + ω×v
ω×v
(0, 1, ω) (0, ω, ω) (0, ω, ω)
(1, ω, ω)

Fig. 1. A PT-net (a) with its sequential (b) and step (c) coverability trees

A step U is *enabled*, at a marking M of $\mathcal{N}$ if ${}^\bullet U \leq M$. Thus, in order for U to be enabled at M, for each place p, the number of tokens in p under M should at least be equal to the accumulated number of tokens needed as input to each of the transitions in U, respecting their multiplicities in U. If U is enabled at M, then it can be *executed* leading to the marking $M' \overset{\text{df}}{=} M - {}^\bullet U + U^\bullet$, denoted $M[U\rangle M'$. Thus the effect of executing U is the accumulated effect of executing each of its transitions (taking into account their multiplicities in U).
A *step sequence* from a marking M to marking M' in $\mathcal{N}$ is a possibly empty sequence $\tau = U_1 \ldots U_n$ of non-empty steps U_i such that $M = M_0\,[U_1\rangle\,M_1\,[U_2\rangle M_2 \cdots M_{n-1}\,[U_n\rangle\,M_n = M'$, for some markings $M_1, \ldots, M_{n-1}$ of $\mathcal{N}$. If τ is a step sequence from M to M' (in $\mathcal{N}$), we write $M\,[\tau\rangle\,M'$ and M' is said to be *step–reachable* or simply *reachable* from M (in $\mathcal{N}$). Obviously, every firing sequence can be seen as a step sequence. Conversely, it is immediate that every step leading from a marking M to M' can be sequentialised to a firing sequence from M to M'. Hence, thanks to this substep property, *fs*–reachability and *step*–reachability are the same for nets.

The net in Figure 1(a) has infinitely many step sequences starting from $(1,0,0)$, e.g., $\tau_1 = t$, $\tau_2 = t(t+v)$, $\tau_3 = t(t+v)(t+2{\times}v)$ and $\tau_4 = t(t+v)(t+2{\times}v)(t+3{\times}v)$.

A *Place/Transition net* (or PT-net) is a net equipped with an initial marking. It is specified as a tuple $\mathcal{N} = (P, T, W, M_0)$, where $\mathcal{N}' = (P, T, W)$ is its underlying net, and M_0 is a marking of $\mathcal{N}'$. All terminology and notation with respect to enabling, firing, and steps carry over from $\mathcal{N}'$ to $\mathcal{N}$. A step (firing) sequence of $\mathcal{N}$ is a step (firing) sequence starting from its initial marking M_0. The set of *reachable* markings of $\mathcal{N}$ consists of all markings reachable from M_0.

A place p of a PT-net $\mathcal{N} = (P, T, W, M_0)$ is *bounded* if there is $n \in \mathbb{N}$ such that $M(p) \leq n$ for every marking M reachable from M_0; otherwise it is *unbounded*. $\mathcal{N}$ itself is *bounded* if all its places are bounded. Considering the PT-net in Figure 1(a), one can easily see that 1 is the only bounded place.

The *place boundedness* problem for PT-nets is to decide whether a given place of a PT-net is bounded; the *boundedness* problem is to decide whether all places in a given PT-net are bounded.

In the subsequent constructions of coverability trees, we use *extended markings* and *extended steps*, generalising the markings and steps defining the execution semantics of PT-nets, to extended multisets of respectively places and transitions. Enabling and firing, as well as the result of executing transitions(s), are defined as for the finite case. Note that since $\omega - \omega = \omega$, an ω-marked place remains ω-marked even after the execution of a step which 'removes' from it ω tokens. It should be stressed that the ω-entries in extended markings and steps do not represent actual tokens or fired transitions, but rather, indicate that the number of tokens or simultaneous firings of transitions can be arbitrarily high.

3 Coverability Tree

We begin by recalling the notion of a coverability tree for a PT-net and how such tree can be constructed assuming the sequential semantics of PT-nets (see, e.g., [10,12,17,7]). A coverability tree $CT = (V, A, \mu, v_0)$ for a PT-net $\mathcal{N} = (P, T, W, M_0)$ has a set of nodes V, a root node v_0, and a set of directed labelled arcs A with labels in T. Each node v is labelled by an extended marking $\mu(v)$ of $\mathcal{N}$. A t-labelled arc from v to w will be denoted as $v \xrightarrow{t} w$. We write $v \leadsto_A w$ (or $v \leadsto_A^{\sigma} w$) to indicate that node w can be reached from node v (with σ as the sequence of labels along the path from v to w).

An algorithm for the construction of CT is given in Table 1. Initially, CT has one node corresponding to the initial marking. A node labelled with an extended marking that already occurs as a label of a processed node, is terminal and does not need to be processed since its successors already appear as successors of this earlier node. For each transition enabled at the extended marking of the node being processed, a new node and a new arc labelled with that transition between these two nodes is added. The label of the new node is the extended marking reached by executing that transition. Note that the algorithm as we give it here is non-deterministic since one may choose which node to process next. Imposing an order on the processing of the nodes is not relevant for the results mentioned here. A key aspect of the algorithm in Table 1 is the condition (*) which allows one to replace some of the integer entries of an extended marking by ω (to indicate that the number of tokens in the corresponding place can be arbitrarily high). This is justified by the monotonicity of the sequential semantics of PT-nets, according to which any sequence of transitions (starting from some M') labelling the path from an ancestor node to a newly generated one (and leading to M with $M' < M$) can be repeated indefinitely. This implies the unboundedness of all places p for which $M'(p) < M(p)$.

The following are well-known facts (see, e.g., [10,7,3]) about the algorithm in Table 1, proving its correctness and indicating how its result CT is a finite representation of the firing sequences of the PT-net $\mathcal{N}$ and provides a useful covering set for its reachable markings. First of all, we observe that the algorithm always terminates. This fact can be proved using condition (*) in Table 1, necessary for the introduction of additional ω-entries in the labels of the nodes.

Fact 1. *CT is finite.* ◇

Table 1. Algorithm generating a coverability tree of a PT-net $\mathcal{N} = (P, T, W, M_0)$

```
$CT = (V, A, \mu, v_0)$ where $V = \{v_0\}$, $A = \varnothing$ and $\mu[v_0] = M_0$
$unprocessed = \{v_0\}$
while $unprocessed \neq \varnothing$
    let $v \in unprocessed$
    if $\mu[v] \notin \mu[V \setminus unprocessed]$ then
    for every $\mu[v][t\rangle M$
        $V = V \uplus \{w\}$ and $A = A \cup \{v \xrightarrow{t} w\}$ and $unprocessed = unprocessed \cup \{w\}$
        if there is $u$ such that $u \rightsquigarrow_A v$ and $\mu[u] < M$                    (*)
        then $\mu[w](p) = ($if $\mu[u](p) < M(p)$ then $\omega$ else $M(p))$
        else $\mu[w] = M$
    $unprocessed = unprocessed \setminus \{v\}$
```

All firing sequences of $\mathcal{N}$ are represented in CT, though sometimes one needs to 'jump' from one node to another (labelled by the same extended marking).

Fact 2. *For each firing sequence $M_0[t_1\rangle M_1 \ldots M_{n-1}[t_n\rangle M_n$ of $\mathcal{N}$, there are arcs $v_0 \xrightarrow{t_1} w_1, v_1 \xrightarrow{t_2} w_2, \ldots, v_{n-1} \xrightarrow{t_n} w_n$ in CT such that: (i) $\mu[w_i] = \mu[v_i]$ for $i = 1, \ldots, n-1$; and (ii) $M_i \leq \mu[v_i]$ for $i = 0, \ldots, n-1$, and $M_n \leq \mu[w_n]$.* ◇

As a consequence, each reachable marking of $\mathcal{N}$ is covered by an extended marking occurring as a label in CT. Also, by the next fact, the ω-entries of an extended marking appearing in CT, indicate that there are reachable markings of $\mathcal{N}$ which simultaneously grow arbitrarily large on the corresponding places and have, for other places, exactly the same entries as the extended marking.

Fact 3. *For every node v of CT and $k \geq 0$, there is a reachable marking M of $\mathcal{N}$ which is a k-approximation of $\mu[v]$, i.e., $M \Subset_k \mu[v]$.* ◇

Consequently, boundedness of (each place of) $\mathcal{N}$ can be read off from CT.

Fact 4. *A place p of $\mathcal{N}$ is bounded* iff *$\mu[v](p) \neq \omega$ for every node v of CT.* ◇

A coverability tree CT for the PT-net in Figure 1(a) is shown in Figure 1(b). Note that according to the facts above, the markings of places 2 and 3 can grow unboundedly at the same time and place 1 is the only bounded place.

4 Coverability Tree and Step Semantics

The construction in Table 1 is satisfactory when one considers the sequential semantics of PT-nets. However, it turns out to be problematic when steps and step sequences are relevant. Consider, for example, the two PT-nets in Figure 2(a, b) for which the algorithm in Table 1 generates the same coverability tree shown in Figure 2(c). Yet, clearly, the first PT-net enables arbitrarily large steps (multiple occurrences of a) whereas the latter enables only singleton steps. An attempt to fix the problem could be to use steps of executed transitions rather than single

Table 2. Algorithm generating a step coverability tree of a PT-net $\mathcal{N} = (P, T, W, M_0)$

```
$SCT = (V, A, \mu, v_0)$ where $V = \{v_0\}$, $A = \varnothing$ and $\mu[v_0] = M_0$
$unprocessed = \{v_0\}$
while $unprocessed \neq \varnothing$
    let $v \in unprocessed$
    if $\mu[v] \notin \mu[V \setminus unprocessed]$ then
    for every $\mu[v][U\rangle M$ with $U \in select(\mu[v])$
        $V = V \uplus \{w\}$ and $A = A \cup \{v \xrightarrow{U} w\}$ and $unprocessed = unprocessed \cup \{w\}$
        if there is $u$ such that $u \rightsquigarrow_A v$ and $\mu[u] < M$                    (**)
        then $\mu[w](p) =$ (if $\mu[u](p) < M(p)$ then $\omega$ else $M(p)$)
        else $\mu[w] = M$
    $unprocessed = unprocessed \setminus \{v\}$
```

transitions to label the arcs. But this still would not be enough since, as in the case of the PT-net in Figure 2(a), there may be infinitely many steps enabled at a reachable extended marking. The solution as presented next is to adapt the coverability tree construction by incorporating not only ordinary steps, but also extended steps with ω-components.

Table 2 shows an algorithm for constructing a step coverability tree. It is similar to that in Table 1 but uses extended steps rather than single transitions to label edges. The `for`-loop is executed for steps from a *finite* yet sufficiently representative subset $select(.)$ of extended steps enabled at the non-empty extended marking under consideration. We define $select(\mu[v])$ as the set of all extended steps of transitions U enabled at $\mu[v]$ with $U(t) = \omega$ for each transition t such that $\omega \times t$ is enabled at $\mu[v]$. We refer to the algorithm resulting from this instantiation as the SCTC (step coverability tree construction). Figures 1(c) and 2(d) show the results of applying SCTC to the nets in Figures 1(a) and 2(a).

We now establish the correctness of the SCTC in Table 2. The proof of the first result is based on the monotonicity of the step semantics employed in condition (**).

Theorem 1 ([13]). *SCT is finite.*

The next result shows that every step sequence of the PT-net can be retraced in SCT if not exactly, then at least through a covering step sequence.

Theorem 2. *For each step sequence $M_0[U_1\rangle \dots [U_n\rangle M_n$ of $\mathcal{N}$, there are arcs $v_0 \xrightarrow{V_1} w_1, v_1 \xrightarrow{V_2} w_2, \dots, v_{n-1} \xrightarrow{V_n} w_n$ in SCT such that: (i) $U_i \leq V_i$ for $i = 1, \dots, n$, and $\mu[w_i] = \mu[v_i]$ for $i = 1, \dots, n-1$; and (ii) $M_i \leq \mu[v_i]$ for $i = 0, \dots, n-1$, and $M_n \leq \mu[w_n]$.*

Proof. We proceed by induction on n. Clearly, the base case for $n = 0$ holds. Assume that the result holds for n and consider $M_n[U_{n+1}\rangle M_{n+1}$.

Let v_n be the first generated node such that $\mu[v_n] = \mu[w_n]$. As $M_n \leq \mu[v_n]$ and $M_n[U_{n+1}\rangle M_{n+1}$, there is M such that $\mu[v_n][U_{n+1}\rangle M$ and $M_{n+1} \leq M$. Let V_{n+1} be the $\leq$-smallest step in $select(\mu[v_n])$ satisfying $U_{n+1} \leq V_{n+1}$ (such a

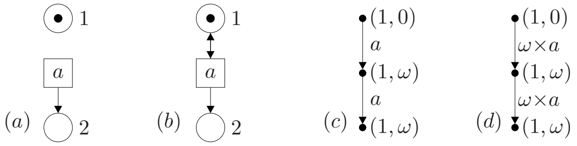


Fig. 2. Two PT-nets (a, b) with their sequential (c) coverability tree, and a step (d) coverability tree of the first PT-net

step always exists). Moreover, let M' be such that $\mu[v_n][V_{n+1}\rangle M'$. It is easy to see that $M_{n+1} \leq M'$. We then observe that during the processing of v_n an arc $v_n \xrightarrow{V_{n+1}} w_{n+1}$ is created such that $M_{n+1} \leq \mu[w_{n+1}]$ which follows from $M_{n+1} \leq M' \leq \mu[w_{n+1}]$. □

As stated next, the ω-entries of the extended markings appearing in SCT, faithfully indicate (simultaneous) unboundedness of the corresponding places.

If μ is a multiset over some set X, then we let $\mu_{\omega\mapsto k}$ denote the multiset over X such that, for all $x \in X$, $\mu_{\omega\mapsto k}(x) = k$ if $\mu(x) = \omega$, and $\mu_{\omega\mapsto k}(x) = \mu(x)$ otherwise.

Theorem 3. *For every node v of SCT and $k \geq 0$, there is a reachable marking M of $\mathcal{N}$ such that $M \Subset_k \mu[v]$.*

Proof. By induction on the distance from the root of the nodes of the tree. In the base case, $v = v_0$ is the root of the tree and so $\mu[v] = M_0$. Suppose that the result holds for a node w, and that $w \xrightarrow{U} v$ with $\mu[w][U\rangle M''$. Note that by the SCTC $dom_\omega(\mu[w]) \subseteq dom_\omega(M'') \subseteq dom_\omega(\mu[v])$. Let $k \in \mathbb{N}$.

First assume that $dom_\omega(M'') = dom_\omega(\mu[v])$. Consider now $Y \overset{\mathrm{df}}{=} U_{\omega\mapsto 0}$ and let M' be a reachable marking of $\mathcal{N}$ such that $M' \Subset_{k'} \mu[w]$ where $k' = k + 1 + |{}^\bullet Y|$. Then $Y \overset{\mathrm{df}}{=} U_{\omega\mapsto 0}$ is enabled at M'. Let M be the reachable marking of $\mathcal{N}$ such that $M'[Y\rangle M$. Since $t \in dom_\omega(U)$ implies that all input and output places of t are in $dom_\omega(\mu[w])$, it now follows immediately that $M \Subset_k \mu[v]$.

Next consider the case that $dom_\omega(M'') \neq dom_\omega(\mu[v])$. Thus $R \overset{\mathrm{df}}{=} \{p \in P \mid \mu[v](p) = \omega \wedge M''(p) \neq \omega\} \neq \varnothing$. For all places p not in R we have $\mu[v](p) = M''(p)$. From the construction of SCT we then know, that there is a node u such that $u \rightsquigarrow_A v$ and $\mu[u] < M''$. Hence there is a path $u = w_1 \xrightarrow{U_1} w_2 \ldots w_n \xrightarrow{U_n} w_{n+1} = v$ (i.e., $w_n = w$ and $U_n = U$) in SCT. Let $W_i \overset{\mathrm{df}}{=} (U_i)_{\omega\mapsto 0}$ for $i = 1, \ldots, n$ and let $cons \overset{\mathrm{df}}{=} \sum_{i\in\{1,\ldots,n\}} |{}^\bullet(W_i)|$ be the total number of tokens consumed along the arcs of the path from u to v by non-ω occurrences of transitions. Let M' be a reachable marking of $\mathcal{N}$ such that $M' \Subset_{k'} \mu[u]$ where $k' = k + 1 + k \cdot cons$. Then $\sigma = (W_1 \ldots W_n)^k$ is enabled at M'. Let M be the reachable marking of $\mathcal{N}$ such that $M'[\sigma\rangle M$. As before, it can now easily be seen that that $M \Subset_k \mu[v]$. □

From Theorems 1, 2, and 3 it follows that step coverability trees (like coverability trees) can be used to decide on the boundedness of places of a PT-net.

Theorem 4. *Place p of $\mathcal{N}$ is bounded* iff $\mu[v](p) \neq \omega$ *for all nodes v of SCT.*

The step coverability tree however makes it possible to investigate concurrency aspects of the behaviour of PT-nets. Not only, as implied by Theorem 3 are all executable steps covered in the SCT (by the labels of the arcs), it also gives exact information on possible unbounded auto-concurrency and potential simultaneous execution of unboundedly many occurrences of (different) transitions. As an example one may compare the step coverability tree in Figure 1(c) derived for the PT-net in Figure 1(a) with its coverability tree in Figure 1(b).

Theorem 5. *For every $k \geq 0$ and every W labelling an arc in SCT, there is a step U enabled at a reachable marking of $\mathcal{N}$ satisfying $U \Subset_k W$.*

Proof. Let $v \xrightarrow{W} w$ be an arc in SCT. Moreover, let $k \in \mathbb{N}$, $U = W_{\omega \mapsto k+1}$, and $k' = |{}^{\bullet}U|$. From W being enabled at $\mu[v]$, it follows that $dom_\omega({}^{\bullet}W) \subseteq dom_\omega(\mu[v])$. By Theorem 3, there is a reachable M of $\mathcal{N}$ such that $M \Subset_{k'} \mu[v]$ and so U is enabled at M. This and $U \Subset_k W$ completes the proof. □

We then obtain a result which, together with Theorem 1, implies that the step executability problem for PT-nets is decidable.

Theorem 6. *A step U is enabled at some reachable marking of $\mathcal{N}$* iff *there is an arc in SCT labelled by W such that $U \leq W$.*

Proof. Follows immediately from Theorems 2 and 5 and the substep property of the step semantics of PT-nets by which a step U is enabled at a marking M whenever $U \leq U'$ (U is a substep of U') for some U' enabled at M. □

5 Weak Monotonicity and Component-Based Systems

When talking about a component-based distributed system, one is typically specifying its *architecture* in terms of components communicating through, e.g., point-to-point buffers or partial broadcast. In many cases, such a *static* description is the only aspect of compositionality which is explicitly specified, and one simply assumes that the *dynamic* behaviour follows a standard execution rule like the one given earlier in this paper in terms of step sequences for PT-nets, or in terms of interleaving sequences (traces) as in the case of standard process algebras [15,11,2,16]. As a result, when dealing with software systems, one might not take into account the fact that an individual component would often run on a dedicated multi-core processor, or that the clocks of some of these processors can be tightly synchronised, where one would therefore expect task schedulers to achieve a significant degree of urgency and synchronisation of enabled tasks within component(s). In case of hardware, similar observations can be made in the context of networks-on-a-chip.

A way to capture such synchronicity is to stipulate that actions belonging to a specific component (or a set of components) are executed with *maximal concurrency*, leading to a clustering of actions into synchronously operating *localities*, and allowing asynchronous execution at the inter-locality level. As a result, the component-based nature of the system is reflected at the level of behaviours.

These considerations led to the introduction, in [14], of *PT-nets with localities* (PTL-nets, for short). Formally, a PTL-net is a PT-net augmented with a *locality mapping* associating localities to transitions, and so partitioning the set of transitions in disjoint sets of co-located transitions. The new enabling condition for steps allows only those steps to occur which are *locally maximally concurrent* with respect to the localities they involve. A special subclass of PTL-nets is one where the locality mapping maps all transitions to a single location leading to what is usually referred to as *maximally concurrent* or *maximal step* semantics. In such a case, we omit the explicit locality mapping and state that a step U is *max-enabled* at a marking M if ${}^\bullet U \leq M$ and there is no transition t such that ${}^\bullet U + {}^\bullet t \leq M$. The notions of *max-reachability*, etc, are defined accordingly.

For the problem considered in this paper, an important feature of the (locally) maximal step semantics is that such steps cannot in general be split and so the interleaving view of their semantics is not accurate. In particular, the set of max-reachable markings of a PT-net (lmax-reachable markings of a PTL-net) is in general a proper (and typically much smaller) subset of the *fs*–reachable markings. Another aspect is that, although the dynamic behaviour of PTL-nets is not monotonic it can be seen as monotonic in a weak sense. Increasing the number of tokens will never invalidate the enabledness of individual transitions. Thus providing more tokens may disable a previously enabled step, but will always lead to the enabling of an extension of this step.

In the last part, we explore the step coverability tree construction for marked graphs [4] subject to the maximally concurrent semantics. Marked graphs do not exhibit local choices between transitions and are typically used to model systems with a high degree of concurrency. A *marked graph* is an *unweighted* PT-net $\mathcal{N} = (P, T, W, M_0)$, i.e., $W(x, y) \leq 1$ for all $x, y \in (T \times P) \cup (P \times T)$, such that each place p has one input and one output transition (i.e., $|{}^\bullet p| = |p^\bullet| = 1$) and each transition t has at least one input and one output place (i.e., $|{}^\bullet t| \geq 1 \leq |t^\bullet|$). Although marked graphs are a simple class of nets, they are practically relevant [24] (see also [9] for a related system model) as they can be thought of as representing systems consisting of (strongly connected) components without local choices which communicate through asynchronous buffers, a fairly common component-based architecture [5].

Each strongly connected component of a marked graph is covered by (concurrent) circuits that may share transitions. A *circuit* (cycle) of $\mathcal{N}$ is a non-empty sequence $x_1, x_2, \ldots, x_k$ of distinct places and transitions such that $x_{i+1} \in {x_i}^\bullet$ for all $1 \leq i \leq k-1$, and $x_1 \in {x_k}^\bullet$. It is well-known (see, e.g., [6] where marked graphs are called T-systems) that the token count on a circuit of a marked graph is invariant under the firing of transitions. As a consequence, transitions with an input place belonging to an initially unmarked circuit, will never be enabled. On the other hand, a marked graph is live if and only if all its circuits have at least one token in the initial marking. (A PT-net is said to be *live* if for all its

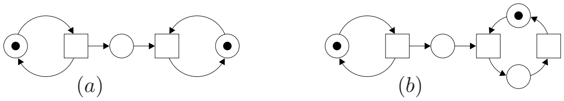


Fig. 3. Two live marked graphs

fs–reachable markings M and transitions t there is a marking *fs*–reachable from M which enables t.) We will restrict ourselves here to live marked graphs.

Figure 3 shows two live marked graphs, which both have an unbounded (buffer) place under the sequential semantics, but under the maximally concurrent semantics only the one on the right has an unbounded place. However, adding one token to the left cycle in Figure 3(a) and one token to the right cycle in Figure 3(b), would reverse the situation. Examples like these demonstrate that boundedness for maximally concurrent marked graphs is a non-trivial problem, being sensitive both to the graph structure and initial marking.

For the class of marked graphs and the maximal step semantics we define the mapping *select*(.) as in the previous section, but now consider only max-enabled steps. Thus $select(\mu[v])$ is the set of all extended steps of transitions U max-enabled at $\mu[v]$ with $U(t) = \omega$ for each transition t such that $\omega \times t$ is enabled at $\mu[v]$. Furthermore we adapt the algorithm in Table 2 by changing line (**) to

'`if` there are u, u' such that $u \rightsquigarrow_A^\tau u' \rightsquigarrow_A^\tau v$ and $\mu[u] < \mu[u'] < M$'

That is, we require not only an inequality on markings, but *double* inequalities and the *same* (maximal) step sequences in-between the three markings. As illustrated by the marked graph in Figure 3(a), the original condition (**) is too weak. Since, in the maximal step semantics the fact that $M[U_1 \dots U_k\rangle M'$ and $M < M'$ does not necessarily guarantee that U_1 is enabled at M', there is no guarantee that $U_1 \dots U_k$ can be repeated indefinitely.

We refer to the result of a run of the thus modified algorithm as $maxSCT_{mg}$. Before establishing that $maxSCT_{mg}$ has the desired properties, we prove using weak monotonicity, as a general property of all unweighted PT-nets executed under the maximal step semantics, that maximal step sequences that do not lead to a decrease of the number of tokens per place and which can be repeated at least twice from a marking can be repeated indefinitely from that marking. In what follows, we use $M\langle\tau\rangle$ to denote the marking reached from a marking M after executing step sequence τ. Moreover, $\#_t(\tau)$ will denote the number of occurrences of any transition t within τ.

Theorem 7. *Let $\mathcal{N}$ be an unweighted PT-net with initial marking M_0. If κ and τ are two sequences of steps such that $\kappa\tau\tau \in steps_{max}(\mathcal{N})$ and $M_0\langle\kappa\rangle \leq M_0\langle\kappa\tau\rangle \leq M_0\langle\kappa\tau\tau\rangle$, then $\kappa\tau^i \in steps_{max}(\mathcal{N})$ for all $i \geq 1$.*

Proof. Let $\tau = U_1 \dots U_k$ and $k \geq 1$. By $M_0\langle\kappa\tau\rangle \leq M_0\langle\kappa\tau\tau\rangle$ and U_1 being enabled at $M_0\langle\kappa\tau\rangle$, there is U such that $U_1 \leq U$ and $\kappa\tau\tau U \in steps_{max}(\mathcal{N})$. Moreover, since $\mathcal{N}$ has only arcs with weight 1 and $M_0\langle\kappa\tau\tau\rangle - M_0\langle\kappa\tau\rangle = M_0\langle\kappa\tau\rangle - M_0\langle\kappa\rangle$ it follows that $U = U_1$. Similarly, we can show that $\kappa\tau\tau U_1 \dots U_i \in steps_{max}(\mathcal{N})$ for every $i \leq k$, and so $\kappa\tau^3 \in steps_{max}(\mathcal{N})$. Moreover,

$M_0\langle\kappa\tau^3\rangle - M_0\langle\kappa\tau^2\rangle = M_0\langle\kappa\tau^2\rangle - M_0\langle\kappa\tau^1\rangle$. Hence $M_0\langle\kappa\tau^2\rangle \leq M_0\langle\kappa\tau^3\rangle$. Proceeding in this way, we easily see that $\kappa\tau^i \in steps_{max}(\mathcal{N})$ for all $i \geq 1$. □

We now define a class of extended marked graphs ($\mathcal{EMG}$) which, intuitively, are strongly connected live marked graphs supplied with some additional infrastructure for (acyclic) communication.

A Each strongly connected live marked graph belongs to $\mathcal{EMG}$.

B Let $\mathcal{N}$ be a net in $\mathcal{EMG}$, t be a transition of $\mathcal{N}$, and p be a fresh place with an arbitrary marking. Adding an arc from t to p results in a net which belongs to $\mathcal{EMG}$.

C Let $\mathcal{N}$ be a net in $\mathcal{EMG}$, $p_1, \ldots, p_k$ ($k \geq 1$) be places of $\mathcal{N}$ without outgoing arcs, t a fresh transition, and $q_1, \ldots, q_m$ ($m \geq 1$) be fresh places with arbitrary markings. Adding an arc from each p_i to t, and from t to each q_j, results in a net which belongs to $\mathcal{EMG}$.

D Let $\mathcal{N}$ be a net in $\mathcal{EMG}$, $p_1, \ldots, p_k$ ($k \geq 1$) be distinct places of $\mathcal{N}$ without outgoing arcs, $\mathcal{N}'$ be another, disjoint, strongly connected live marked graph and $t_1, \ldots, t_k$ be distinct transitions of $\mathcal{N}'$. Adding an arc from each p_i to t_i results in a net which belongs to $\mathcal{EMG}$.

Properties important here are that each live marked graph belongs to $\mathcal{EMG}$ and that each PT-net in $\mathcal{EMG}$ is live. The next two results provide some insight in the dynamics of the component nets. By the first observation, the firing distance between transitions in a component of a marked graph is always bounded.

Proposition 1. *Let $\mathcal{N}$ be a strongly connected marked graph. Then there is a constant ℓ such that $|\#_t(\tau) - \#_u(\tau)| \leq \ell$ for every step sequence τ of $\mathcal{N}$ and for all transitions t and u.*

Proof. Clearly, for any two transitions on any given circuit there is such a constant. The result follows from this, $\mathcal{N}$ being connected and covered by circuits which synchronise on common transitions, and the inequality $|a - b| \leq |a - c| + |c - b|$ for any a, b, c. □

Secondly, when a component returns to a marking it must be the case that each of its transitions has fired the same number of times.

Proposition 2. *Let $\mathcal{N}$ be a strongly connected marked graph with initial marking M_0, and $\kappa\tau$ be a step sequence of $\mathcal{N}$ such that $M_0\langle\kappa\rangle = M_0\langle\kappa\tau\rangle$. Then $\#_t(\tau) = \#_u(\tau)$ for all transitions t and u.*

Proof. The equality holds if t and u belong to a circuit. Moreover, $\mathcal{N}$ is connected and covered by circuits which synchronise on common transitions. □

Now we are ready for a precise characterisation of the behaviour of (extended) marked graphs subject to the maximal step semantics.

Theorem 8. *Let EMG be a net in $\mathcal{EMG}$ with initial marking M_0.*
Then there are non-empty sequences of non-empty steps, κ and τ, such that $\kappa\tau^i \in steps_{max}(EMG)$ for all $i \geq 1$. Moreover, $\#_t(\tau) > 0$ for all t.

Proof. We proceed by induction on the structure of EMG.

A: Then EMG is finite state, and for each reachable marking there is only one maximal step enabled. Since EMG is live, the statement follows.

B: Then p does not have any influence on the behaviour of $\mathcal{N}$ within EMG, and the result holds by the induction hypothesis.

C: By the induction hypothesis, there are non-empty sequences of non-empty steps, κ and τ, such that $\kappa\tau^i \in \mathit{steps}_{max}(\mathcal{N})$ for all $i \geq 1$. Moreover, $\#_{t'}(\tau) > 0$ for all t' of $\mathcal{N}$. We observe that t does not have any influence on the behaviour of $\mathcal{N}$ within EMG. Hence there are sequences μ_i ($i \geq 0$) of steps such that: $\mu_0\mu_1 \ldots \mu_i \in \mathit{steps}_{max}(EMG)$ for all $i \geq 0$ with $\widehat{\mu_0} = \kappa$ and $\widehat{\mu_i} = \tau$ ($i \geq 1$), where each $\widehat{\mu_j}$ is μ_j after deleting all the occurrences of t.

Let $M_i \overset{\mathrm{df}}{=} M_0\langle\mu_0\mu_1 \ldots \mu_i\rangle$ for $i \geq 1$. We observe that from the definition of the maximal step semantics it follows that, for every $h \leq k$ and $m \geq 1$,

$M_m(p_h) = M_1(p_h) + m \cdot \#_{t_h}(\tau) - \min_j\{M_1(p_j) + (m-1) \cdot \#_{t_j}(\tau) + \#_{t_j}(\tau')\}$

where each t_i is the input transition of p_i and $\tau = \tau'U$ with U a step of EMG. Hence, for sufficiently large m, the minimum is realised by l such that $\#_{t_l}(\tau) = \min_j\{\#_{t_j}(\tau)\} > 0$ and, moreover, $M_1(p_l) + \#_{t_1}(\tau') \leq M_1(p_j) + \#_{t_j}(\tau')$ for all j such that $\#_{t_l}(\tau) = \#_{t_j}(\tau)$. As a result, for such an m, $M_m(p_l) = U(t_l)$. This means, in turn, that there is L such that $M(p_l) \leq L$ for every marking M max-reachable from M_m (we can take $L = U(t_l) + \#_{t_1}(\tau)$). Furthermore, there is $n \geq m$ such that for all j satisfying $\#_{t_l}(\tau) < \#_{t_j}(\tau)$ and marking M max-reachable from M_n, we have $M(p_j) > L$ and so all such places are irrelevant for the executability of t at M (i.e., they cannot block it). Finally, for each j satisfying $\#_{t_1}(\tau) = \#_{t_j}(\tau)$, we have that $M_i(p_j) = M_1(p_j) - M_1(p_l) + U(t_l)$ for all $i \geq m$. Hence it follows from the definition of the maximal step semantics that $\widetilde{\mu_m} = \widetilde{\mu_{m+1}} = \ldots$, where each $\widetilde{\mu_j}$ is μ_j with all the occurrences of transitions of $\mathcal{N}$ deleted, and so $\mu_m = \mu_{m+1} = \ldots$ which yields the desired result since also $\#_t(\mu_m) > 0$.

D: By the induction hypothesis, there are non-empty sequences of non-empty steps, κ and τ, such that $\kappa\tau^i \in \mathit{steps}_{max}(\mathcal{N})$ for all $i \geq 1$. Moreover, $\#_{t'}(\tau) > 0$ for all t' of $\mathcal{N}$. We observe that $\mathcal{N}'$ does not have any influence on the behaviour of $\mathcal{N}$ within EMG. Hence there are sequences μ_i ($i \geq 0$) of steps such that: $\mu_0\mu_1 \ldots \mu_i \in \mathit{steps}_{max}(EMG)$ for all $i \geq 0$ with $\widehat{\mu_0} = \kappa$ and $\widehat{\mu_i} = \tau$ ($i \geq 1$), where each $\widehat{\mu_j}$ is μ_j after deleting all the occurrences of transitions in $\mathcal{N}'$.

Let $M_i \overset{\mathrm{df}}{=} M_0[\mu_0\mu_1 \ldots \mu_i\rangle$ for $i \geq 1$. Moreover, let u_i be the only input transition of p_i, and $I \overset{\mathrm{df}}{=} \{p_i \mid \forall j : \#_{u_i}(\tau) \leq \#_{u_j}(\tau)\}$.

We first observe that, due to Proposition 1 and $\mathcal{N}'$ being finite state, there is m such that for each marking M which is max-reachable from M_m, places $p_j \notin I$ have no influence on the firing of t_j.

Moreover, also by Proposition 1, for some $n \geq m$, there is K such that for all $p, p' \in I$ and all markings M max-reachable from M_m, $|M(p) - M(p')| \leq K$. Now, if no place $p \in I$ ever blocks a transition belonging to $\mathcal{N}'$ after M_h, for some $h \geq n$, then $\mathcal{N}'$ behaves under the maximal step semantics as a strongly connected live marked graph within EMG. By the induction hypothesis, it has its own sequences of steps κ' and τ' as in the formulation of this result, and after at most $|\tau| \cdot |\tau'|$ steps $\mathcal{N}$ and $\mathcal{N}'$ start executing a common τ''.

Otherwise we have such blocking infinitely many times, and so there is L such that for infinitely many j we have $M_j(p) \leq L$ for some $p \in I$. And so there is Q such that for all such indices j, $M_j(p) \leq Q$ for all $p \in I$. It therefore follows that the same marking on places I is repeated infinitely many times on the places of I as well as the places of $\mathcal{N}'$ (recall that $\mathcal{N}'$ is finite-state). This means that we can find two markings, M' and after that M'', for which, in addition, we have that they happened after the execution of the same U_l within τ on the part of $\mathcal{N}$. It now suffices to consider τ' to be the step sequence between M' and M'', and then proceed similarly as in the previous case. □

That $maxSCT_{mg}$ is finite now follows from Theorem 8 and the fact that live marked graphs under the under maximal concurrent semantics are deterministic systems (the reachability graph does not have any branching nodes). Moreover, that the ω-markings generated by $maxSCT_{mg}$ are sound and indeed reflect unboundedness of places follows from Theorem 7.

Actually, Theorem 8 provides us with a complete description of the behaviour of extended marked graphs. Since each extended marked graph EMG is a deterministic system (with a 'linear' reachability graph), it follows from our result above that there are κ and τ, such that every maximal step sequence of EMG is a prefix of $\kappa\tau^i$ for some $i \geq 0$. Thus, intuitively, Theorem 8 states that, under maximal concurrency, a live marked graph behaves as a set of cogs (each cog corresponding to the cyclic behaviour of a strongly connected component) which initially can progress fairly erratically, but which after some time all synchronise and work in a highly regular manner, irrespective of the initial marking (provided that it puts at least one token on any circuit). This has the consequence that if one embeds the marked graph in an environment which can add or delete tokens (without emptying any circuit) then the system sooner or later self-stabilises assuming a cyclic pattern of execution (see also [9] for comparable results).

6 Concluding Remarks

In this paper we have continued our investigation initiated in [13] on the construction of step coverability trees which can be useful for the analysis of various Petri net models. It can be already said that SCTs extend in a smooth way the standard CTs and can be used to answer concurrency-related questions to which the latter are simply insensitive. We also added results on the viability of the SCT construction presented in [13] for inhibitor nets by adapting the construction to deal with marked graphs executed under the maximally concurrent semantics. In particular, the results allow one to decide (place) boundedness for such a system model. Although the class of marked graphs is limited, we feel that the results we obtained are a crucial stepping stone in the discovery of coverability tree constructions for wider classes of system models.

Acknowledgement. We thank the reviewers for their constructive comments. This research was partially supported by the Rae&Epsrc Davac project, and Nsfc Grant 60433010.

References

1. Agerwala, T.: A Complete Model for Representing the Coordination of Asynchronous Processes. In: Hopkins Comp. Research Rep., vol. 32. Johns Hopkins Univ. (1974)
2. Baeten, J.C.M., Weijland, W.P.: Process algebra. Cambridge Tracts in Theoretical Computer Science, vol. 18. Cambridge University Press, Cambridge (1990)
3. Busi, N.: Analysis Issues in Petri Nets with Inhibitor Arcs. Theoretical Computer Science 275, 127–177 (2002)
4. Commoner, F., Holt, A.W., Even, S., Pnueli, A.: Marked Directed Graphs. J. Comput. Syst. Sci. 5, 511–523 (1971)
5. Darondeau, P., Genest, B., Thiagarajan, P.S., Yang, S.: Quasi-Static Scheduling of Communicating Tasks. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 310–324. Springer, Heidelberg (2008)
6. Desel, J., Esparza, J.: Free Choice Nets. Cambridge University Press, Cambridge (1995)
7. Desel, J., Reisig, W.: Place/Transition Petri Nets. In: [20], pp. 122–173
8. Esparza, J., Nielsen, M.: Decidability Issues for Petri Nets: A Survey. J. of Inf. Processing and Cybernetics 30, 143–160 (1994)
9. Ghamarian, A.H., Geilen, M.C.W., Basten, T., Theelen, B.D., Mousavi, M.R., Stuijk, S.: Liveness and Boundedness of Synchronous Data Flow Graphs. In: FMCAD 2006, pp. 12–16 (2006)
10. Hack, M.: Decision Problems for Petri Nets and Vector Addition Systems. Technical Memo 59, Project MAC, MIT (1975)
11. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
12. Karp, R.M., Miller, R.E.: Parallel Program Schemata. J. Comput. Syst. Sci. 3, 147–195 (1969)
13. Kleijn, J., Koutny, M.: Steps and Coverability in Inhibitor Nets. In: Lodaya, K., Mukund, M., Ramanujam, R. (eds.) Perspectives in Concurrency Theory, pp. 264–295. Universities Press, Hyderabad (2008)
14. Kleijn, J., Koutny, M., Rozenberg, G.: Process Semantics for Membrane Systems. J. of Aut., Lang. and Comb. 11, 321–340 (2006)
15. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
16. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes. Inf. and Comp. 100, 1–77 (1992)
17. Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice-Hall, Englewood Cliffs (1981)
18. Petri, C.A.: Fundamentals of a Theory of Asynchronous Information Flow. In: IFIP Congress 1962, pp. 386–390. North Holland, Amsterdam (1962)
19. Rozenberg, G., Engelfriet, J.: Elementary Net Systems. In: [20], 12–121
20. Reisig, W., Rozenberg, G. (eds.): APN 1998. LNCS, vol. 1491. Springer, Heidelberg (1998)
21. Rozenberg, G., Thiagarajan, P.S.: Petri Nets: Basic Notions, Structure, Behaviour. In: Rozenberg, G., de Bakker, J.W., de Roever, W.-P. (eds.) Current Trends in Concurrency. LNCS, vol. 224, pp. 585–668. Springer, Heidelberg (1986)
22. Valmari, A.: The State Explosion Problem. In: [20], 429–528
23. Wimmel, H.: Entscheidbarkeitsfragen bei Petri Netzen. Habilitation (2007)
24. Yoeli, M.: Specification and Verification of Asynchronous Circuits using Marked Graphs. In: Rozenberg, G. (ed.) APN 1987. LNCS, vol. 266, pp. 605–622. Springer, Heidelberg (1987)

Program Logics for Sequential Higher-Order Control

Martin Berger

Department of Informatics, University of Sussex

Abstract. We introduce a Hoare logic for call-by-value higher-order functional languages with control operators such as `callcc`. The key idea is to build the assertion language and proof rules around an explicit logical representation of jumps and their dual 'places-to-jump-to'. This enables the assertion language to capture precisely the intensional and extensional effects of jumping by internalising rely/guarantee reasoning, leading to simple proof rules for higher-order functions with `callcc`. We show that the logic can reason easily about non-trivial uses of `callcc`. The logic matches exactly with the operational semantics of the target language (observational completeness), is relatively complete in Cook's sense and allows efficient generation of characteristic formulae.

1 Introduction

Non-trivial control manipulation is an important part of advanced programming and shows up in many variants such as jumps, exceptions and continuations. Research on axiomatic accounts of control manipulation starts with [10], where a simple, imperative first-order low-level language with `goto` is investigated. Recently, this research tradition was revived by a sequence of works on similar languages [2–4, 7, 24, 29, 32, 34]. None of those investigates the interplay between advanced control constructs and higher-order features. The present paper fills this gap and proposes a logic for ML-like call-by-value functional languages with advanced control operators (`callcc`, `throw`). The key difficulty in axiomatising higher-order control constructs for functional languages (henceforth "higher-order control") is that program logics are traditionally based on the idea of abstracting behaviour in terms of input/output relations. This is a powerful abstraction for simple languages but does not cater well for jumping, a rather more intensional form of behaviour. Consider the well-known program $\mathtt{argfc} \overset{def}{=} \mathtt{callcc}\ \lambda k.(\mathtt{throw}\ k\ \lambda x.(\mathtt{throw}\ k\ \lambda y.x))$ [12]. This function normalises to a λ-abstraction, but, as [28] investigates, distinguishes programs by application that are indistinguishable in the absence of continuations: $(\lambda x.(x\ 1);(x\ 2))\ \mathtt{argfc} = 1$ and $(\lambda x.\lambda y.(x\ 1);(y\ 2))\ \mathtt{argfc}\ \mathtt{argfc} = 2$ with $M;N$ being the sequential composition of M and N, binding more tightly than λ-abstraction. The reason is that continuations carry information about contexts that may be returned (jumped) to later. Thus, values in languages with higher-order control are no longer simple entities, precluding logics based on input/output relations. Two ways of dealing with the intensional nature of control manipulation suggest themselves:

- Using continuation-passing style (CPS) transforms [33] to translate away control manipulating operators and then reason about transformed programs in logics like [16] for functional languages.

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 194–211, 2010.

- Using a direct syntactic representation of intensional features.

We choose the second option for pragmatic reasons: It is difficult to reconstruct a program's specification from the specification of its CPS transform. This is because CPS transforms increase the size of programs, even where higher-order control is not used. This increases reasoning complexity considerably. In contrast, in our approach programs or program parts that do not feature higher-order control can be reasoned about in simpler logics for conventional functional languages. The more heavyweight logical apparatus for higher-order control is required only where control is manipulated explicitly, leading to more concise proofs and specifications.

Key Elements of the Present Approach. This work makes three key proposals for a logical treatment of higher-order control.

- *Names* as an explicit representation of places to jump to, or being jumped to.
- *Jumps* $\overline{x}\langle\tilde{e}\rangle A$ as an explicit logical operator which says that a program jumps to x carrying a vector $\tilde{e}$ of values, and after jumping, A holds. Jumps are complementary to the evaluation formulae $x\langle\tilde{e}\rangle A$, studied in [5, 16, 18, 36], which means a jump to x carrying values $\tilde{e}$ leads to a program state where A holds.
- *Rely/guarantee formulae* $\{A\}B$ and *tensor* $A \circ B$. $\{A\}B$ says that if the environment is as specified by A, then the program together with the environment will act as constrained by B. Likewise, $A \circ B$ says a program has a part that is as described by A, and a part that is as given by B. Rely/guarantee formulae generalise implication, and tensor generalise conjunction because in e.g. $A \wedge (B \supset C)$ a free variable must have the same type in A as in B and C. With rely/guarantee formulae, we weaken this requirement: e.g. in $\overline{x}\langle 2u\rangle \wedge \{x\langle vw\rangle \overline{w}\langle v+1\rangle\}\overline{u}\langle 3\rangle$ the variable x is used to output in the left conjunct and for input in the right conjunct, with the input occurring in the rely part of the rely/guarantee formula. The left conjunct says that the program jumps to x carrying 2 and u (an intensional specification at x). The right conjunct says that if the environment offers a function to be invoked at x that computes the successor of its first argument and returns the result at the second, then the jump to u carrying 3 will happen, a more extensional specification in that the program together with the assumed environment behaves as a function. Similarly, $\overline{x}\langle 3\rangle \circ x\langle 3\rangle A$ uses x with different polarities, specifying a program that contains a jump to x carrying 3 to a target for this jump.

Informal Explanation. Operationally, a program, for example the constant 5, can be seen as a value-passing jump carrying 5 to some distinguished name, called *default port*, left implicit in the language, but made explicit in implementations, usually as a return address on the stack. It can be left implicit in the absence of higher-order control because there are no alternatives for returning: every function, if it returns at all, does so at the default port. Higher-order control breaks this simplicity: for example `throw` k 5 will jump to k and not to the default port. Our logic deals with multiple return points by *naming* the default port in judgements, giving rise to the following shape of judgements (for total and partial correctness):

$$M :_{\overline{u}} A$$

It asserts that the program M satisfies the formula A, assuming that M's default port is named u (we do not need preconditions because they can be simulated using rely/guarantee formulae, see below). Using explicit jumps we can specify:

$$5 :_{\overline{u}} \overline{u}\langle 5\rangle \qquad\qquad \texttt{throw}\, k\, 5 :_{\overline{u}} \overline{k}\langle 5\rangle.$$

The left-hand side says that the program 5 terminates and jumps to the default port u carrying 5 as a value. The assertion on the right expresses that $\texttt{throw}\, k\, 5$ also terminates with a jump carrying 5, but now the jump is to k, which is not the default port.

Evaluation formulae are used to specify the behaviour of functions. When a function like $\lambda x.x+1$ is invoked, the result is returned at the default port of the invocation. As functions can be invoked more than once and in different contexts (in the present context, invoking a function f is the same as jumping to f, and we use both phrases interchangeably), different default ports are needed for different invocations. In implementations, a dynamically determined place on the stack is used for this purpose. In addition, when a λ-abstraction like $\lambda x.x+1$ is evaluated, the λ-abstraction itself, i.e. $\lambda x.x+1$, is returned at its default port. To express such behaviour we use the following specification (writing $\overline{u}(a)A$ for $\exists a.\overline{u}\langle a\rangle A$, and $a(xm)A$ for $\forall xm.a\langle xm\rangle A$).

$$\lambda x.x+1 :_{\overline{u}} \overline{u}(a)a(xm)\overline{m}\langle x+1\rangle$$

This judgement states that the abstraction returns a name a at the default port. This name can be jumped *to* (i.e. invoked) with two arguments, a number x and a name m, the default port for invocations of a. If invoked, the successor of x will be returned at m.

The role of rely/guarantee formulae is to generalise and internalise preconditions. Consider the application $g\,3$. If jumps to g with two arguments, a number x and a return port u, yield a jump $\overline{u}\langle x+x\rangle$ then the evaluation of $g\,3$ with default port u should induce a jump $\overline{u}\langle 6\rangle$. In a program logic with preconditions, we would expect to be able to derive $\{g(xm)\overline{m}\langle x+x\rangle\}\ g\,3\ :_{\overline{u}} \{\overline{u}\langle 6\rangle\}$. With rely/guarantee formulae we can express this by defining

$$\{A\}\, M\ :_{\overline{m}} \{B\} \qquad \overset{def}{=} \qquad M :_{\overline{m}}\ \{A\}B.$$

The advantage of internalising preconditions with rely/guarantee formulae are threefold. (1) Key structural relationships between jumps and evaluation formulae are easily expressible as axioms like: $\overline{x}\langle\tilde{e}\rangle\ \supset\ \{x\langle\tilde{e}\rangle A\}A$. It states that e.g. a jump $\overline{g}\langle 3u\rangle$ makes A true whenever the environment guarantees that jumps *to* g with arguments 3 and u will validate A. (2) We gain more flexibility in localising and manipulating assumptions, leading to more succinct and compositional reasoning. To see why, consider a complicated formula $C[\overline{x}\langle 2\rangle]$ containing a jump to x. Using the axiom just given, and setting $A \overset{def}{=} x\langle 2\rangle\overline{u}\langle 3\rangle$, we know that $\overline{x}\langle 2\rangle \supset \{A\}\overline{u}\langle 3\rangle$, hence $C[\overline{x}\langle 2\rangle]$ implies $C[\{A\}\overline{u}\langle 3\rangle]$. Such reasoning is cumbersome if all assumptions have to be concentrated in the precondition. Moreover, local hypotheses can be collected, i.e. we can usually infer from $C[\{A\}\overline{u}\langle 3\rangle]$ to $\{A\}C[\overline{u}\langle 3\rangle]$, hence conventional reasoning based on rigid pre-/postconditions remains valid unmodified without additional cost (all relevant rules and axioms are derivable). The added fluidity in manipulating assumptions is vital for reasoning about involved forms of mutually recursive jumping. (3) Finally, the most important virtue of internalising preconditions is sheer expressive power. With rely/guarantee formulae, it easy to use different assumptions in a single formula: consider $A \stackrel{def}{=} g(xm)\overline{m}\langle x+x\rangle$ and $B \stackrel{def}{=} g(xm)\overline{m}\langle x \cdot x\rangle$. We can now specify $g\ 3 :_{\overline{u}}\ (\{A\}\overline{u}\langle 6\rangle) \wedge \{B\}\overline{u}\langle 9\rangle$. This expressiveness enables convenient reasoning about complicated behavioural properties of programs that would be difficult to carry out otherwise.

Contributions. The present work provides the first general assertion method with compositional proof rules for higher-order functions with functional control (`callcc` and similar operators) and recursion under the full type hierarchy. The work identifies as key ingredients in this approach: (1) An explicit representation of jumps in formulae, which can specify intensional aspects of control operators in a uniform manner. (2) Rely/guarantee formulae and an associated tensor to facilitate local specification of extensional as well as intensional aspects of higher-order control, and to enable complicated forms of reasoning not otherwise possible. (3) Proof rules and axioms that capture the semantics of PCF^{+}precisely, as demonstrated by strong completeness results. Missing proofs can be found in the full version of this paper.

2 PCF with Jumps

Now we define our programming language. We extend PCF with `callcc` and `throw`, and call the resulting language PCF^{+}. Arguments are evaluated using *call-by-value* (CBV). Later we briefly consider μPCF, a variant of CBV PCF with different control operators. The relationship between both is explained in [21]. *Types*, *terms* and *values* are given by the grammar below. Sums, products and recursive types for PCF^{+} are straightforward and are discussed in the full version of this paper.

$$\alpha ::= \mathbb{N} \mid \mathbb{B} \mid \mathsf{Unit} \mid \alpha \to \beta \mid (\alpha)^{?} \qquad V ::= x \mid \mathsf{c} \mid \lambda x^{\alpha}.M \mid \mathtt{rec}\, f^{\alpha}.\lambda x^{\beta}.M$$
$$M ::= V \mid MN \mid \mathsf{op}(\tilde{M}) \mid \mathtt{if}\, M\, \mathtt{then}\, N\, \mathtt{else}\, N' \mid \mathtt{callcc} \mid \mathtt{throw}$$

Here $(\alpha)^{?}$ corresponds to $(\alpha\ \mathsf{cont})$ in SML and is the type of continuations with final answer type α, c ranges over constants like $0,1,2,...$, op over functions like addition. We write e.g. $ab3$ for the vector $\langle a,b,3\rangle$, $\tilde{M}$ for the vector $\langle M_0,...,M_{n-1}\rangle$, etc; $x,f,...$ range over *variables*. Names are variables that can be used for jumping. The notions of *free variables* $\mathsf{fv}(M)$ and *bound variables* $\mathsf{bv}(M)$ of M are defined as usual. Typing judgements $\Gamma \vdash M : \alpha$ are standard, with Γ being a finite, partial map from variables to the types α. *From now on we assume all occurring programs to be well-typed.* The semantics of PCF^{+} is straightforward, cf. [28].

3 The Logic

This section defines the syntax and semantics of the logic. Since variables in programs are typed, and the logic speaks about such variables, our logic is typed, too. Types are those of PCF^{+}, with two generalisations. (1) We add a type $(\tilde{\alpha})^{!}$ which is the type *being-jumped-to* with arguments typed by the vector $\tilde{\alpha}$. (2) We no longer need function spaces, because e.g. $\alpha \stackrel{def}{=} \mathbb{N} \to \mathbb{B}$ can now be decomposed into $\alpha^{\circ} \stackrel{def}{=} (\mathbb{N}(\mathbb{B})^{?})^{!}$. Type α° holds of names that can be *jumped to* with two arguments, first a number and then another name, which might be used for subsequent jumps carrying a boolean. This is the behaviour of functions $\mathbb{N} \rightarrow \mathbb{B}$ under call-by-value evaluation. If we denote by $\overline{\alpha}$ the result of changing all occurring ? in α into ! and vice versa, and if we denote by α° the result of translating PCF^+ types as just described, then:

$$(\alpha \rightarrow \beta)^\circ = (\overline{\alpha^\circ}(\beta^\circ)^?)^!.$$

Our types are given by the grammar:

$$\alpha ::= \mathbb{N} \mid \mathbb{B} \mid \mathsf{Unit} \mid (\alpha)^? \mid (\alpha\beta)^? \mid (\alpha)^! \mid (\alpha\beta)^!$$

Types play essentially the same role in our logic as they do in programming languages, namely to prevent terms that do not make sense, like $x = 5 + \mathsf{t}$ or $\overline{x}\langle 3 \rangle \circ x\langle\rangle A$. Since our use of types is straightforward, the reader can mostly ignore types in the remainder of the text, as long as he or she bears in mind that *all occurring formulae and judgements must be well-typed*. (Further information about this typing system is given in [15].)

Expressions, Formulae, Assertions. Expressions are standard ($e ::= x \mid \mathsf{c} \mid \mathsf{op}(\tilde{e})$) and formulae for our logic are generated by the following grammar.

$$A ::= e = e' \mid A \wedge B \mid \neg A \mid \forall x^\alpha.A \mid x\langle\tilde{e}\rangle A \mid \overline{x}\langle\tilde{e}\rangle A \mid \{A\}B \mid A \circ B$$

Variables, constants and functions are those of §2. Standard logical operators such as T implication and existential quantification are defined as usual. We often omit type annotations. Logical operators have the usual rules of precedence, e.g. $\forall x.A \wedge B$ should be read as $\forall x.(A \wedge B)$, $A \circ B \wedge C$ as $(A \circ B) \wedge C$, and $\{A\}B \wedge C$ is short for $(\{A\}B) \wedge C$. We write $\mathsf{fv}(A)$ for A's free variables, and $A^{\text{-}x}$ indicates that $x \notin \mathsf{fv}(A)$. Names are also variables. Typing environments, $\Gamma, \Delta, \ldots$ are defined as finite maps from names to types. Typing judgements for expressions $\Delta \vdash e : \alpha$ and formulae $\Delta \vdash A$ are defined as usual, e.g. x must be of type $\mathbb{N}$ in $x + 3 = 2$. The new operators are typed as follows.

- $\Gamma \vdash \overline{x}\langle\tilde{e}\rangle A$ if $\Gamma \vdash x : (\tilde{\alpha})^?$, $\Gamma \vdash e_i : \beta_i$, $\beta_i \in \{\alpha_i, \overline{\alpha_i}\}$ and $\Gamma \vdash A$.
- $\Gamma \vdash x\langle\tilde{e}\rangle A$ if $\Gamma \vdash x : (\tilde{\alpha})^!$, $\Gamma \vdash e_i : \beta_i$, $\beta_i \in \{\alpha_i, \overline{\alpha_i}\}$ and $\Gamma \vdash A$.
- For rely/guarantee formulae $\Gamma \vdash \{A\}B$ we say x is *compensated* in A if the type of x in A is dual to that in Γ. For example $\overline{x}\langle 2y \rangle \supset \{x\langle 2y \rangle \overline{y}\langle 3 \rangle\}B$ is typable under $\Delta \overset{def}{=} x : (\mathbb{N}(\mathbb{B})^!)^?, y : (\mathbb{B})^?$.

We write e.g. $\overline{x}\langle y \cdot \rangle A$ to stand for any $\overline{x}\langle yz \rangle A$ such that z is fresh and does not occur in A, and likewise for evaluation formulae. We write $\overline{x}\langle \tilde{e}(y) \rangle A$ for $\exists y.\overline{x}\langle \tilde{e}y \rangle A$, assuming y not to occur in $\tilde{e}$. *Judgements*, also called *assertions*, are of the form $M :_{\overline{m}} A$. Judgements must be well-typed, i.e. M and A must be well-typed and the variables common to A and M must be given consistent types, e.g. $g\ 3 :_{\overline{u}} \{g\langle 4u \rangle \mathsf{T}\}2 = 3$ is well-typed, but $g\ 3 :_{\overline{u}} \{g\langle z \rangle \mathsf{T}\}2 = 3$ is not.

Examples of Assertions. We continue with simple examples of assertions.

- Let $A \overset{def}{=} g(xk)(\mathsf{even}(x) \supset \overline{k}(a)\mathsf{even}(a))$. This first example specifies a place g to jump to. If a jump to g happens carrying an even number x as first argument and k, the default port, then that invocation at g will return at its default port, carrying another even number. A does not specifying anything if x is odd.

- Next consider the following formulae. $A \overset{def}{=} x(kr)(\overline{k}\langle 7\rangle \vee \overline{r}\langle 8\rangle)$ and $B \overset{def}{=} \{A\}\overline{u}(m)(m = 7 \vee m = 8)$ A specifies a place x to jump to with two arguments, k and r (the default port), both of which are used for jumping: either jumping to k carrying 7, or jumping to the default port carrying 8. B specifies a jump to u carrying 7 or 8, provided the environment provides a place to jump to at x, as just described by A.
- Now consider the formula $A \overset{def}{=} x(ab)\overline{a}\langle bb\rangle$. It says that if we jump to x carrying two arguments, a and b, both being used for jumping, then the invocation at x replies with a jump to a, carrying b twice. Figure 2 shows that $\overline{u}(x)A$ specifies the behaviour of `callcc`, assuming u as default port.
- Finally, consider the following formula. $A \overset{def}{=} \overline{n}(b)b(xy)\overline{n}(c)c(zr)\overline{r}\langle x\rangle$. The formula A specifies a jump to n, carrying a function b that can be jumped to with two arguments, x and y. Of those, y is subsequently ignored. If b is invoked, it jumps to n again, carrying another function c, which also takes two arguments, z and r. Of these z is also ignored, but r is jumped to immediately, carrying x. It can be shown that A specifies `argfc`.

Models and the Satisfaction Relation. This section sketches key facts about the semantics of our logic and states soundness and completeness results. We use a typed π-calculus to construct our semantics. This choice simplifies models and reasoning about models for the following reasons.

- Models and the satisfaction relation need to be built only once and then cater for many different languages with functional control like PCF and μPCF. Thus soundness of axioms needs to be proven only once. Proving soundness and completeness is also simpler with π-calculus based models because powerful reasoning tools are available, e.g. labelled transitions, that languages with higher-order sequential control currently lack.
- Using processes, the semantics is simple, intuitive and understandable, as well as capturing behaviour of higher-order control precisely. The typed processes that interpret PCF$^+$ or μPCF-programs are up to bisimilarity exactly the morphisms (strategies) in the control categories that give fully abstract models to PCF$^+$ or μPCF[13,21]. Hence the present choice of model gives a direct link with game-based analysis of control.

Processes. The grammar below defines processes with expressions e as above, cf. [17] for details.

$$P ::= 0 \mid \overline{x}\langle \tilde{e}\rangle \mid\, !x(\tilde{v}).P \mid (\nu x)P \mid P|Q \mid \texttt{if}\; e \;\texttt{then}\; P \;\texttt{else}\; Q$$

We can use this calculus to give fully abstract encodings of PCF$^+$ and μPCF [17,21]. Translation is straightforward and we show some key cases.

$$\begin{array}{llll} [\![\lambda x.M]\!]_u \overset{def}{=} \overline{u}(a)!a(xm).[\![M]\!]_m & & [\![\texttt{throw}]\!]_u \overset{def}{=} \overline{u}(a)!a(xm)\overline{m}(b)!b(y\cdot)\overline{x}\langle y\rangle \\ [\![MN]\!]_u \overset{def}{=} (\nu m)([\![M]\!]_m|!m(a).(\nu n)([\![N]\!]_n|!n(b).\overline{a}\langle bu\rangle)) & & [\![\texttt{callcc}]\!]_u \overset{def}{=} \overline{u}(a)!a(xm).\overline{x}\langle mm\rangle \end{array}$$

This translation generalises a well-known CPS transform [33]. All cases of the translation are syntactically essentially identical with the corresponding logical rules. This simplifies soundness and completeness proofs and was a vital rule-discovery heuristic.

The Model and Satisfaction Relations. *Models of type* Γ are of the form (P,ξ) where P is a process and ξ maps values names and variables to their denotation. We write $\models M :_{\overline{m}} A$ if for all appropriately typed-models (P,ξ) with m fresh in ξ we have

$$([\![M]\!]_m \xi | P, \xi) \models A$$

This satisfaction relation works for total and partial correctness, since termination can be stated explicitly through jumps in total correctness judgements. On formulae, the satisfaction relation is standard except in the following four cases, simplified to streamline the presentation (here $\cong$ is the contextual congruence on typed processes).

- $(P,\xi) \models \overline{x}\langle y\rangle$ if $P \cong Q|\overline{a}\langle b\rangle$, $\xi(x) = a, \xi(y) = b$.
- $(P,\xi) \models x\langle y\rangle A$ if $P \cong Q|!a(v).R$ with $\xi(x) = a$ and $(P|\overline{a}\langle\xi(y)\rangle,\xi) \models A$.
- $(P,\xi) \models \{A\}B$ if for all Q of appropriate type $(Q,\xi) \models A$ implies $(P|Q,\xi) \models B$.
- $(P,\xi) \models A \circ B$ if we can find Q,R such that $P \cong Q|R$, $(Q,\xi) \models A$ and $(R,\xi) \models B$.

The construction shows that rely/guarantee formulae correspond to (hypothetical) parallel composition [20].

4 Axioms and Rules

This section introduces all rules and some key axioms of the logic. We start with the latter and concentrate on axioms for jumps, tensor and rely/guarantee formulae. Some axioms correspond closely to similar axioms for implication and conjunction. *All axioms and rules are included in the logic exactly when they are typable.*

Axioms for Dynamics. We start with the two axioms that embody the computational dynamics of jumping. The first expresses the tight relationship between jumping and being-jumped-to (evaluation formulae):

$$\overline{u}\langle\tilde{e}\rangle A \circ u\langle\tilde{e}\rangle B \qquad \supset \qquad A \circ B \qquad\qquad (\text{CUT})$$

[CUT] says that if a system is ready to make a jump to u, say it satisfies $\overline{u}\langle\tilde{e}\rangle A$, and if the system also contains the target for jumps to u, i.e. it satisfies $u\langle\tilde{e}\rangle B$, then that jump will happen, and $A \circ B$ will also be true of the system.

The next axiom says that a jump $\overline{x}\langle\tilde{e}\rangle A$ which guarantees A implies the weaker statement that if the environment can be jumped to at x with arguments $\tilde{e}$, then B holds, provided the environment can rely on A in its environment.

$$\overline{x}\langle\tilde{e}\rangle A \quad \supset \quad \{x\langle\tilde{e}\rangle\{A\}B\}B \qquad\qquad (\text{XCHANGE})$$

Further Axioms for Tensor and Rely/Guarantee Formulae. Now we present some axioms for the tensor that show its close relationship with conjunction. In parallel, we also exhibit axioms for rely/guarantee formulae that relate them with implication. As before, we assume that both sides of an entailment or equivalence are typed under the same typing environment. This assumption is vital for soundness, as we illustrate below.

$$\begin{array}{ccc} A \circ B \equiv A \wedge B & A \circ B \supset A & A \supset \{B\}A \\ A \circ (B \circ C) \equiv (A \circ B) \circ A & (\forall x.A) \circ B^{\text{-}x} \equiv \forall x.(A \circ B) & \{A\}\{B\}C \equiv \{A \circ B\}C \\ A \circ B \equiv B \circ A & \{A\}B \equiv A \supset B & B \circ \{B\}A \supset A \end{array}$$

Our explanation of these axioms starts on the left. The first axiom says that if $A \wedge B$ are typable then tensor is just conjunction. This does not imply that $\overline{x}\langle 3\rangle \circ x\langle 3\rangle A$ is equivalent to $\overline{x}\langle 3\rangle \wedge x\langle 3\rangle A$, since $\overline{x}\langle 3\rangle \wedge x\langle 3\rangle A$ is not typable. However $(x = 3 \circ y = 1) \equiv (x = 3 \wedge y = 1)$ is valid. The next two axioms below state associativity and commutativity of tensor. The top axiom in the middle shows that tensor is not like parallel composition, because the tensor can "forget" their component formulae. The axiom below shows that tensor associates as expected with quantification. The bottom axiom in the middle shows that rely/guarantee formulae reduce to implication if all free variables have the same type in A as in B, i.e. $(\{x\langle\tilde{e}\rangle\overline{a}\}\overline{x}\langle e\rangle) \equiv ((x\langle\tilde{e}\rangle\overline{a}) \supset \overline{x}\langle e\rangle)$ is not a valid instance of the axiom, but $(\{x\langle\tilde{e}\rangle\overline{a}\}x\langle\tilde{e}\rangle\overline{b}) \equiv ((x\langle\tilde{e}\rangle\overline{a}) \supset x\langle\tilde{e}\rangle\overline{b})$ is. The top right axiom shows that it is possible to weaken with a rely formula. The middle axiom on the right shows how to merge two assumptions in rely/guarantee formulae. The bottom right axiom can be seen as a typed form of Modus Ponens, and we call it [MP]. The expected forms of weakening also hold, i.e. if $A \supset A'$ then $A \circ B$ implies $A' \circ B$, $\{A'\}B$ implies $\{A\}B$ and $\{B\}A$ implies $\{B\}A'$.

Further Axioms for Jumps and Evaluation Formulae. Before moving on to rules, we present some axioms for jumps and evaluation formulae.

$$\begin{array}{rr} x\langle\tilde{e}\rangle(A \wedge y\langle\tilde{g}\rangle B) \equiv y\langle\tilde{g}\rangle(B \wedge x\langle\tilde{e}\rangle A) & x\langle\tilde{e}\rangle\mathsf{T} \equiv \mathsf{T} \\ A \circ (\overline{x}\langle\tilde{e}\rangle B) \equiv \overline{x}\langle\tilde{e}\rangle(A \circ B) & \overline{x}\langle\tilde{e}\rangle \wedge \overline{y}\langle\tilde{g}\rangle \supset (x = y \wedge \tilde{e} = \tilde{g}) \end{array}$$

The top left axiom states that free variables like x and y that can be jumped to, are 'always there', i.e. they cannot come into existence only after some function has been invoked. The top right axiom says that places to jump to cannot 'refuse' arguments: in other words, the statement $x\langle e\rangle\mathsf{T}$ carries no information. This axiom is called [NOINFO]. The bottom left axiom says that if a program contains a part that jumps at x then the program as a whole can also jump at x, provided that the program does not contain a component that offers an input at x (not offering an input at x is implicit in typability of the axiom). Finally, the last axiom expresses that our language is sequential: at most one jump can happen at any time.

Rules for PCF$^+$. The total correctness rules for PCF$^+$ are given in Figure 1. Rules are subject to straightforward well-formedness conditions. *From now on we assume all rules to be well-typed.* We explain the rules in some detail. As [VAR, CONST, ABS] have already been sketched in the introduction, we start with the rule for application. The purpose of [APP], the rule for function application, is to ensure the coordination of functions and their invocations by jumps. One issue is the generation and management of default ports: the present approach requires that a (terminating) function application *may* return its result at the application's default port, assuming the evaluations of the

$$\frac{M :_{\overline{m}} A}{\lambda x.M :_{\overline{u}} \overline{u}(a)a(xm)A}\ \text{Abs} \qquad \frac{\lambda x.M :_{\overline{u}} \overline{u}(a)A}{\mathtt{rec}\, g.\lambda x.M :_{\overline{u}} \overline{u}(a)\exists g.(\mathsf{fw}_{ga} \circ A)}\ \text{Rec} \qquad \frac{-}{\mathtt{c} :_{\overline{u}} \overline{u}\langle \mathtt{c}\rangle}\ \text{Const}$$

$$\frac{M :_{\overline{m}} A \quad N :_{\overline{n}} B}{MN :_{\overline{u}} \exists m.(A \circ m(a)\exists n.(B \circ n(b)\overline{a}\langle bu\rangle))}\ \text{App} \qquad \frac{-}{\mathtt{callcc} :_{\overline{u}} \overline{u}(a)a(xm)\overline{x}\langle mm\rangle}\ \text{Ccc} \qquad \frac{-}{x :_{\overline{u}} \overline{u}\langle x\rangle}\ \text{Var}$$

$$\frac{-}{\mathtt{throw} :_{\overline{u}} \overline{u}(a)a(xm)\overline{m}(b)b(y\cdot)\overline{x}\langle y\rangle}\ \text{Throw} \qquad \frac{M :_{\overline{m}} A \quad N :_{\overline{u}} B}{M+N :_{\overline{u}} \exists m.(A \circ m(a)\exists n.(B \circ n(b)\overline{u}\langle a+b\rangle))}\ \text{Add}$$

$$\frac{M :_{\overline{m}} A \quad N :_{\overline{u}} B \quad N' :_{\overline{u}} C}{\mathtt{if}\, M\, \mathtt{then}\, N\, \mathtt{else}\, N' :_{\overline{u}} \exists m.(A \circ m(a)((a = \mathsf{t} \supset B) \wedge (a = \mathsf{f} \supset C)))}\ \text{If} \qquad \frac{M :_{\overline{u}} A \quad A \supset B}{M :_{\overline{u}} B}\ \text{Cons}$$

Fig. 1. Total Correctness rules for PCF$^+$. The *forwarder* is given by $\mathsf{fw}_{xy} \stackrel{def}{=} x(\tilde{v})\overline{y}\langle\tilde{v}\rangle$.

function itself, and that of the argument return their respective results at (distinct) default ports themselves. [App] achieves this by explicitly representing the sequence of jumps that are integral parts of evaluating a function application. First the jump to the default port of the function is received by an evaluation formula at m. It receives an argument a. Then the evaluation of the argument is triggered, and its result, should it return at the fresh default port n, is received by a second evaluation formula at n. Finally, should both, the function and its argument return at their respective default ports, a jump to a carrying b and the application's default port u is executed. By typing we know that the jump to a must find an evaluation formula expecting two arguments.

Why do we have to represent the internals of application evaluation in the logic explicitly, rather then have them implicit as in the simpler logics for PCF [16]? After all, even in PCF, these jumps take place, albeit behind the scenes. The answer is that because of continuations, functions can return more than once, i.e. can jump to their default port more than once. The function `argfc` from the introduction is an example of such behaviour. The axiomatisation of PCF in [16] hides default ports, because programs cannot return anywhere but at default ports. It might not be possible to give a logical account of returning to a port more than once without explicit representation of default ports.

Representing jumps and default ports in a single formula, as we do in [App], has ramifications for typing: when names (like m, n above) are used in a formula for both, jumping, and for being-jumped-to we need to mediate, in a controlled way, the rigidity of typing, that enforces all names to be used under the same typing. Our rules use tensor for this purpose. All rules can be stated without tensors using just rely/guarantee formulae, but, it seems, not without a making the inference system more complicated.

Using [App], setting $A \stackrel{def}{=} \exists m.((\overline{m}(a)a(xu)\overline{u}\langle x+1\rangle) \circ m(a)\exists n.(\overline{n}\langle 7\rangle \circ n(b)\overline{a}\langle bu\rangle))$ and assuming that $\lambda x.x+1 :_{\overline{m}} \overline{m}(a)a(xr)\overline{r}\langle x+1\rangle$, we infer:

$$\begin{array}{ll}
1 & \lambda x.x+1 :_{\overline{m}} \overline{m}(a)a(xr)\overline{r}\langle x+1\rangle \\ \hline
2 & 7 :_{\overline{n}} \overline{n}\langle 7\rangle \text{Const} \\ \hline
3 & (\lambda x.x+1)7 :_{\overline{u}} A \text{App}, 1, 2
\end{array}$$

The expected judgement $(\lambda x.x+1)7 :_{\overline{u}} \overline{u}\langle 8\rangle$, is by [CONS] and the following implication :

$$\begin{array}{llll} A & \supset \exists a.((a(xu)\overline{u}\langle x+1\rangle)\circ\exists n.(\overline{n}\langle 7\rangle\circ n(b)\overline{a}\langle bu\rangle)) & \supset & \exists a.((a(xu)\overline{u}\langle x+1\rangle)\circ\exists n.\overline{a}\langle 7u\rangle) \\ & \supset \exists a.((a(xu)\overline{u}\langle x+1\rangle)\circ\overline{a}\langle 7u\rangle) & \supset & \overline{u}\langle 8\rangle \end{array}$$

This implication follows from [CUT] and simple logical manipulations.

As second example we consider the application $g\ x$, with an assumption on the behaviour of g. The intent is to illuminate the use of rely/guarantee formulae and the [XCHANGE] axiom. Let $A \overset{def}{=} \mathsf{even}(x)\wedge g(xk)(\mathsf{even}(x)\supset\overline{k}(a)\mathsf{even}(a))$. We want to show that

$$\{A\}\ gx\ :_{\overline{u}} \{\overline{u}(a)\mathsf{even}(a)\}, \qquad (1)$$

recalling that $\{B\}\ M\ :_{\overline{m}} \{C\}$ is short for $M :_{\overline{m}}\ \{A\}B$. First we reason as follows.

1 $g :_{\overline{m}}\ \overline{m}\langle g\rangle$VAR

2 $x :_{\overline{n}}\ \overline{n}\langle x\rangle$VAR

3 $gx :_{\overline{u}}\ \exists m.(\overline{m}\langle f\rangle\circ m(a)\exists n.(\overline{n}\langle x\rangle\circ n(b)\overline{a}\langle bu\rangle))$APP, 1, 2

4 $\{A\}\ gx\ :_{\overline{u}} \{\overline{u}(a)\mathsf{even}(a)\}$.CONS, 3

The interesting step is the last, where we reason as follows.

$$\begin{array}{lll} \exists m.(\overline{m}\langle g\rangle\circ m(a)\exists n.(\overline{n}\langle x\rangle\circ n(b)\overline{a}\langle bu\rangle)) & \supset \exists m.(\overline{m}\langle g\rangle\circ m(a)\exists n.(\overline{a}\langle xu\rangle)) & \supset \\ \exists m.(\overline{m}\langle g\rangle\circ m(a)\overline{a}\langle xu\rangle) & \supset \exists m.\overline{g}\langle xu\rangle & \supset \overline{g}\langle xu\rangle \end{array}$$

The first and third inferences use [CUT], the two others remove unused quantifiers. Theorem 1 shows that $\overline{g}\langle xu\rangle$ is an optimal specification for our program in the sense that anything that can be said at all about the program gx with anchor u can be derived from $\overline{g}\langle xu\rangle$. We continue by deriving (1), using $B \overset{def}{=} \mathsf{even}(x)\supset\overline{u}(a)\mathsf{even}(a)$.

$$\begin{array}{l} \overline{g}\langle xu\rangle \supset \{g\langle xu\rangle(\mathsf{even}(x)\wedge B)\}(\mathsf{even}(x)\wedge B) \supset \{g(xu)(\mathsf{even}(x)\wedge B)\}(\mathsf{even}(x)\wedge B) \\ \qquad \supset \{A\}\overline{u}(a)\mathsf{even}(a) \end{array}$$

The first implication is by [XCHANGE], the others are straightforward strengthening of the precondition, and simple first-order logic manipulations. Now (1) follows by the consequence rule.

The derivation above has a clear 2-phase structure: first a general assertion about the behaviour of the application is derived without assumptions on free variables. Then such assumptions are added using [XCHANGE] and the consequence rule. It is noteworthy that the first phase is mechanical by induction on the syntax of the program, while the second phase takes place without reference to the program. It is possible to use a more traditional style of reasoning, where applications of languages rules and [CONS] are mixed, but this tends to make inferences longer.

Like the rule for application, [REC] is an adaption of the corresponding rule in [16], but forwarding all jumps to the recursion variable g directly to the recursive function at a. This forwarding corresponds to "copy-cat strategies" in game-semantics [1, 19], here realising the feedback loop of jumps to f into a that enables recursion by using tensor. [REC] implies a more convenient rule, given as follows.

$$\frac{\lambda x.M :_{\overline{m}} \overline{m}(a)\forall j \lneq i.\{A[g/a][j/i]\}A}{\texttt{rec}\ g.\lambda x.M :_{\overline{m}} \overline{m}(a)\forall i.A}\ \text{REC'}$$

As first example of using [REC] we consider a simple function $\omega \overset{def}{=} \texttt{rec}\ g.\lambda x.gx$ that diverges upon invocation. Since our rules and axioms are for total correctness, we should not be able to specify anything about ω, except that it terminates and returns at its default port when evaluated as an abstraction, i.e. we show: $\omega :_{\overline{u}} \overline{u}(a)a(xu)\mathsf{T}$. Mechanically we infer the following judgement

$$\omega :_{\overline{u}} \overline{u}(a)\exists g.(\mathsf{fw}_{ga} \circ a(xk)\overline{g}\langle xk\rangle)$$

We use axiomatic reasoning to obtain $\omega :_{\overline{u}} \overline{u}(a)a(xu)\mathsf{T}$ by [CONS].

$$\begin{array}{ll}\overline{u}(a)\exists g.(\mathsf{fw}_{ga} \circ a(xk)\overline{g}\langle xk\rangle) & \supset \overline{u}(a)\exists g.(\mathsf{fw}_{ga} \circ a(xk)\{g\langle xk\rangle\mathsf{T}\}\mathsf{T}) \supset \\ \overline{u}(a)\exists g.(\mathsf{fw}_{ga} \circ \{g\langle xk\rangle\mathsf{T}\}a(xk)\mathsf{T}) & \supset \overline{u}(a)\exists g.(\mathsf{fw}_{ga} \circ \{\mathsf{fw}_{ga}\}a(xk)\mathsf{T}) \quad\ \supset \\ \overline{u}(a)\exists g.a(xk)\mathsf{T} & \supset \overline{u}(a)a(xk)\mathsf{T}\end{array}$$

The first line uses [XCHANGE], the next pushes the assumption of the rely/guarantee formula to the left of the evaluation formula. Then we simply replace that assumption by fw_{ga}. We can do this, because that strengthens the assumption, i.e. weakens the rely/guarantee formula. Then we apply [MP]. The last line removes the superfluous quantifier. We note that there is a simpler derivation of the same fact, relying on the implications:

$$\overline{u}(a)\exists g.(\mathsf{fw}_{ga} \circ a(xk)\overline{g}\langle xk\rangle) \quad \supset \quad \overline{u}(a)\mathsf{T} \quad \supset \quad \overline{u}(a)a(xk)\mathsf{T}.$$

The first of those is just weakening of the tensor, while the second is an instance of [NOINFO].

[CCC] says that `callcc` is a constant, always terminating, and returning at the default port, carrying a function, denoted a, as value. This function takes two arguments, x, the name of another function, and m, the default port for the invocation of a. By typing we know that m must be a function invoked with an argument of continuation type $(\alpha)^?$. Whenever a is invoked, it jumps to x, carrying its default port m as first and second argument. In other words, if the invocation at x terminates at its default port, it does so at a's default port. Moreover, x can also jump to m explicitly. Note that m is duplicated [CCC], i.e. used non-linearly. This non-linearity is the reason for the expressive power of functional control.

We consider another example of reasoning about `callcc`: $M \overset{def}{=} \texttt{callcc}\ \lambda k.7$. Mechanically, we derive

$$M :_{\overline{u}} \underbrace{\exists m.(\overline{m}(a)a(xr)\overline{x}\langle rr\rangle \circ m(a)\exists n.(\overline{n}(b)b(ks)\overline{s}\langle 7\rangle \circ n(b)\overline{a}\langle bu\rangle))}_{A}$$

Then we use axiomatic reasoning to reach the expected judgement $M :_{\overline{u}} \ \overline{u}\langle 7\rangle$.

$$\begin{array}{lll} A \supset \exists a.(a(xr)\overline{x}\langle rr\rangle \circ \exists b.(b(ks)\overline{s}\langle 7\rangle \circ \overline{a}\langle bu\rangle)) & \supset \exists ab.(a(xr)\overline{x}\langle rr\rangle \circ \overline{a}\langle bu\rangle \circ b(ks)\overline{s}\langle 7\rangle) & \\ \quad \supset \exists ab.(\overline{b}\langle uu\rangle \circ b(ks)\overline{s}\langle 7\rangle) & \supset \exists ab.\overline{u}\langle 7\rangle & \supset \overline{u}\langle 7\rangle \end{array}$$

[Throw] says that `throw` is a function returning at its default port a function a which takes x as its first argument (by typing a continuation $(\alpha)^{?}$), and returns at its default port m a second function b, which in turn takes two argument, the first of which is y (of type α). The second argument, the default port of y is ignored, since x will be *jumped* carrying y as argument.

We continue with reasoning about simple programs with `throw`. We show that:

$$\mathtt{throw}\, k\, 3 :_{\overline{u}} \ \overline{k}\langle 3\rangle \qquad\qquad \omega(\mathtt{throw}\, k\, 3) :_{\overline{u}} \ \overline{k}\langle 3\rangle.$$

We begin with the assertion on the left. The assertion for this program will be quite sizable because [App] must be applied twice. The following abbreviation is useful to shorten specifications arising from [App].

$$A \mid_{mnu} B \qquad \overset{def}{=} \qquad \exists m.(A \circ m(a)\exists n.(B \circ n(b)\overline{a}\langle bu\rangle)).$$

Here we assume that u, n do not occur in M and u, m are not in N. We let $\mid_{mnu}$ bind less tightly than all the other operators of the logic. This abbreviation is interesting because of the following derived rule, which is immediate from the rules.

$$\overline{m}(a)a(bu)A \mid_{mnu} \overline{n}(b)B \qquad \supset \qquad \exists ab.(A \wedge B). \tag{2}$$

From [Throw], $k :_{\overline{b}} \ \overline{b}\langle k\rangle$ and $3 :_{\overline{n}} \ \overline{n}\langle 3\rangle$ we get:

$$\mathtt{throw}\, k\, 3 :_{\overline{u}} \ (\overline{g}(a)a(xm)\overline{m}(b)b(y\cdot)\overline{x}\langle y\rangle) \mid_{gbm} \overline{b}\langle k\rangle \mid_{mnu} \overline{n}\langle 3\rangle$$

which simplifies to `throw` k 3 $:_{\overline{u}} \ \overline{k}\langle 3\rangle$ by applying (2) twice. Now we deal with $\omega($`throw` k $3)$. As before: $\omega($`throw` k $3) :_{\overline{u}} \ A$ with $A \overset{def}{=} \overline{m}(a)a(bu)\mathsf{T} \mid_{mnu} \overline{k}\langle 3\rangle$, but we cannot apply (2) since `throw` k 3 does not return at the default port. Instead we reason from the axioms.

$$\exists n.(\overline{k}\langle 3\rangle \circ n(b)\overline{a}\langle bu\rangle) \ \supset \ \exists n.\overline{k}\langle 3\rangle(\mathsf{T} \circ n(b)\overline{a}\langle bu\rangle) \supset \overline{k}\langle 3\rangle\exists n.(\mathsf{T} \circ n(b)\overline{a}\langle bu\rangle) \ \supset \ \overline{k}\langle 3\rangle$$

Here the first line is an application of [Cut], the second switches quantification with a jump, and the third line is by [NoInfo], in addition to straightforward logical manipulations. Thus we can use [Cut] once more and infer:

$$\begin{array}{lll} \overline{m}(a)a(bu)\mathsf{T} \mid_{mnu} \overline{k}\langle 3\rangle & \supset \exists m.((\overline{m}(a).a(bu)\mathsf{T}) \circ m(a)\overline{k}\langle 3\rangle) & \supset \exists mb.((a(bu)\mathsf{T}) \circ \overline{k}\langle 3\rangle) \\ & \supset \overline{k}\langle 3\rangle\exists mb.((a(bu)\mathsf{T}) \circ \mathsf{T}) & \supset \overline{k}\langle 3\rangle \end{array}$$

[If] simply adds a recipient for the default port at M, the condition of the conditional, where a boolean b is received. Depending on b, the specification of one of the branches is enabled. [Add] is similar to [App] and the [Cons], the rule of consequence, is standard in program logics.

A Comment on the Shape of Rules. Program logics are usually presented "bottom-up", meaning that postconditions in the conclusion of rules are just a meta-variable standing for arbitrary (well-typed) formulae. This facilitates reasoning starting from a desired postcondition of the program under specification, and then trying to find an appropriate premise. We have chosen the "top-down" presentation because it gives simpler and more intuitive rules, and shortens inferences substantially. A "bottom-up"presentation of proof rules is possible, and may be useful in some cases. The status of the "bottom-up" rules (e.g. completeness) is yet to be established.

Completeness. A goal of axiomatic semantics is to be in harmony with the corresponding operational semantics. That means that two programs should be contextually indistinguishable if and only if they satisfy the same formulae. This property is called *observational completeness*. We establish observational completeness as a consequence of *descriptive completeness*.

Definition 1. *By $\sqsubseteq$ we mean the standard typed contextual precongurence for* PCF^+*, i.e. $M \sqsubseteq N$ if for $C[M] \Downarrow$ implies $C[N] \Downarrow$ for all closing contexts $C[\cdot]$, where $\Downarrow$ means termination.*

Theorem 1. *(Descriptive Completeness for Total Correctness) Our logic is* descriptively complete*: for all closed M, N (typable under the same typing), A and m, we have: $\vdash M :_{\overline{m}} A$ implies that (1) $\models M :_{\overline{m}} A$ and (2) whenever $\models N :_{\overline{m}} A$ then $M \sqsubseteq N$.*

The proof of this theorem, and the derivation of observational completeness (as well as relative completeness in the sense of Cook) from descriptive completeness follows [14].

The λμ-Calculus. From the rules and axioms for PCF^+, it is easy to derive a logic for $\mu\mathsf{PCF}$, an extension of the λμ-calculus, a Curry-Howard correspondence for classical logic, with a recursion operator. The logic enjoys similar completeness properties.

$$\frac{M :_m A \quad N :_u B}{M+N :_u \exists mn.(A \wedge B \wedge u = a+b)}\ \textsc{SAdd} \qquad \frac{M :_m A}{\lambda x.M :_u u\langle x\rangle_m A}\ \textsc{SAbs} \qquad \frac{M :_{\overline{m}} A}{\lambda x.M :_u u\langle xm\rangle A}\ \textsc{SAbs'}$$

$$\frac{\lambda x.M :_u A}{\texttt{rec}\ g.\lambda x.M :_u \exists g.(\mathsf{fw}_{ga} \circ A)}\ \textsc{SRec} \qquad \frac{M :_m A \quad N :_n B}{MN :_{\overline{u}} \exists mn.((A \wedge B) \circ \overline{m}\langle nu\rangle)}\ \textsc{SApp} \qquad \frac{M :_m m\langle n\rangle_u A \quad N :_n B}{MN :_u \exists mn.(A \wedge B)}\ \textsc{SApp'}$$

$$\frac{-}{\texttt{callcc} :_u u(xm)\overline{x}\langle mm\rangle}\ \textsc{SCcc} \qquad \frac{-}{\texttt{throw} :_u u\langle x\rangle_m m(y\cdot)\overline{x}\langle y\rangle}\ \textsc{SThrow} \qquad \frac{-}{x :_u u = x}\ \textsc{SVar}$$

$$\frac{-}{\texttt{c} :_u u = \texttt{c}}\ \textsc{SConst} \qquad \frac{M :_m A \quad N :_u B \quad N' :_u C}{\texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ N' :_u \exists m.(A \circ (m = \texttt{t} \supset B) \wedge (a = \mathsf{f} \supset C))}\ \textsc{SIf}$$

$$\frac{M :_u A \quad A \supset B}{M :_u B}\ \textsc{SCons} \qquad \frac{M :_{\overline{n}} \exists \tilde{a}.(m\langle e\rangle_u A \circ \overline{m}\langle en\rangle B)}{M :_u \exists \tilde{a}.(A \circ B)}\ \textsc{SCut}$$

Fig. 2. Some derived rules that are useful for reasoning about PCF^+ programs that return at their default port

5 Simplifying Reasoning

PCF-terms are a subset of PCF$^+$-terms. Reasoning about PCF-terms using the logic for PCF$^+$ is moderately more laborious than using a logic tailor-made for PCF like [16]. This is because intermediary jumps in function application are represented explicitly in the former, but not the latter. Reasoning in §4 about simple programs like $(\lambda x.x+1)7$ and `throw` k 3 suggest that intermediate jumps can be eliminated mechanically in applications where a function and its argument return at the default port. We formalise this intuition and obtain simplified derivable logical rules and axioms, that can be used to reason about a large class of programs, including PCF$^+$ programs that do use functional control. We start by defining two syntactic shorthands that apply only to judgements and evaluation formulae that return at their default ports (u fresh in both):

$$M :_m A \stackrel{def}{=} M :_{\overline{u}} \overline{u}(m)A \qquad x\langle\tilde{e}\rangle_m A \stackrel{def}{=} \forall u.x\langle\tilde{e}u\rangle\overline{u}(m)A$$

We write $x(\tilde{y})_m A$ for $\forall\tilde{y}.x\langle\tilde{y}\rangle_m A$. Using this syntax, $\lambda x.x+1$ has the following specification, as we shall show below. $\lambda x.x+1 :_u u\langle x\rangle_m\, m = x+1$. In order to derive specifications like this more efficiently than by expansion of abbreviations, we introduce *derivable* rules and axioms that work directly with this new syntax. Figure 2 lists some rules. Axioms can be simplified in the same way.

Termination at default ports is not the only place where higher-level rules are useful. Examples in §4 indicate that reasoning about non-default jumps also often follows more high-level patterns. To support this intuition, we add more shorthands.

$$M \nearrow A \stackrel{def}{=} M :_{\overline{u}} A \wedge \overline{m}\langle\cdot\rangle \wedge m \neq u \qquad a \bullet e \nearrow \{A\} \stackrel{def}{=} a\langle eu\rangle(A \wedge \overline{m}\langle\cdot\rangle \wedge m \neq u)$$

In both u must be fresh. Rules using these additional rules can be found in Figure 3.

Theorem 2. *All rules in Figures 2 and 3, and all associated axioms are derivable.*

We continue with some further examples of using the derived rules and axioms. We start by deriving $3+$`throw` $k\ 7 :_{\overline{u}} \overline{k}\langle 3\rangle$ once more.

1	$k :_{\overline{n}} \overline{n}\langle k\rangle$	Var
2	$7 :_h h = 7$	SVar
3	`throw` $k\ 7 \nearrow \overline{k}\langle y\rangle$	JThrow”
4	$3 :_m m = 3$	SConst
5	$3+$`throw` $k\ 7 \nearrow \overline{k}\langle y\rangle$	JAdd’

Now we consider an example that show that the simplified rules are also useful when reasoning about programs with free variables. Consider

$$\texttt{callcc}\ x :_m \{A\}(m = 7 \vee m = 8) \tag{3}$$

where $A \stackrel{def}{=} x(kr)(\overline{k}\langle 7\rangle \vee \overline{r}\langle 8\rangle)$. Mechanically, using the simplified rules, we infer

1	$\texttt{callcc} :_a a(bc)\overline{b}\langle cc\rangle$	SVAR
2	$x :_b b = x$	SCCC
3	$\texttt{callcc}\, x :_{\overline{u}} \exists ab.(a(bc)\overline{b}\langle cc\rangle \wedge b = x) \circ \overline{a}\langle bu\rangle$	SAPP, 1, 2
4	$\texttt{callcc}\, x :_{\overline{u}} \overline{x}\langle uu\rangle$	CONS, 3
5	$\texttt{callcc}\, x :_m \{A\}(m = 7 \vee m = 8)$	CONS, 4

Line 4 is by a straightforward application of [CUT] and some straightforward logical manipulations. To get Line 5, we reason as follows.

$$\overline{x}\langle uu\rangle \supset \{x\langle uu\rangle(\overline{u}\langle 7\rangle \vee \overline{u}\langle 8\rangle)\}(\overline{u}\langle 7\rangle \vee \overline{u}\langle 8\rangle) \supset \{A\}(\overline{u}\langle 7\rangle \vee \overline{u}\langle 8\rangle) \supset \overline{u}(m)\{A\}(\overline{m}\langle 7\rangle \vee \overline{m}\langle 8\rangle)$$

The first of these implications uses [XCHANGE], while the second strengthens the precondition of the rely/guarantee formula.

Example (3) shows how easily we can reason about programs that have free variables which are assumed to act like throwing a continuation. Just as easily one can assume that a variable acts like `callcc` and prove $x\ \lambda k.\texttt{throw}\ k\ 7 :_m \{A\}m = 7$, where $A \stackrel{def}{=} x(ab)\overline{a}\langle bb\rangle$.

$$\frac{M \nearrow A}{\lambda x.M :_u u \bullet x \nearrow \{A\}}\ \textsc{JAbs} \qquad \frac{M :_m A \quad N \nearrow B}{M + N \nearrow B}\ \textsc{JAdd'} \qquad \frac{M \nearrow A}{M + N \nearrow A}\ \textsc{JAdd} \qquad \frac{M :_m A \quad N \nearrow B}{MN \nearrow B}\ \textsc{JApp'}$$

$$\frac{M \nearrow A}{\texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ N' \nearrow A}\ \textsc{JAdd} \qquad \frac{M \nearrow A}{\texttt{callcc}\ M \nearrow A}\ \textsc{JCcc} \qquad \frac{M \nearrow A}{MN \nearrow A}\ \textsc{JApp} \qquad \frac{M \nearrow A}{\texttt{throw}\ M\ N \nearrow A}\ \textsc{JThrow}$$

$$\frac{M :_m A \quad N \nearrow B}{\texttt{throw}\ M\ N \nearrow B}\ \textsc{JThrow'} \qquad \frac{M :_{\overline{m}} \overline{m}\langle k\rangle \quad N :_n A}{\texttt{throw}\ M\ N \nearrow \overline{k}(n)A}\ \textsc{JThrow''}$$

Fig. 3. Some derived rules, helpful for reasoning about PCF$^+$ programs that jump

Relating the Logics for PCF and PCF$^+$. The derivable rules and axioms just discussed pose the question of the systematic relationship between the present logic and that for PCF [14, 16]. We give an answer by providing a simple translation of formulae and judgements from the logic for PCF to that for PCF$^+$, and then showing that the inclusion on programs preserves derivability. The idea behind the translation is straightforward: just add fresh default ports.

We continue with a summary of the logic for PCF in [14, 16]. Types and formulae are given by the following grammar, with expressions being unchanged.

$$\alpha ::= \mathbb{N} \mid \mathbb{B} \mid \mathsf{Unit} \mid \alpha \to \beta \qquad\qquad A ::= e = e' \mid A \wedge B \mid \neg A \mid \forall x^{\alpha}.A \mid x\langle e\rangle_y A$$

Judgements are of the form $\{A\}\,M\ :_m \{B\}$. Next is the translation of PCF-formulae into PCF$^+$-formulae.

$$\begin{array}{lll}
\ulcorner e = e' \urcorner \overset{def}{=} e = e' & \ulcorner A \wedge B \urcorner \overset{def}{=} \ulcorner A \urcorner \wedge \ulcorner B \urcorner & \ulcorner \neg A \urcorner \overset{def}{=} \neg \ulcorner A \urcorner \\
\ulcorner \forall x^{\alpha}.A \urcorner \overset{def}{=} \forall x^{\alpha^\circ}.\ulcorner A \urcorner & \ulcorner x\langle e \rangle_y A \urcorner \overset{def}{=} \forall u.x\langle eu \rangle \overline{u}(y) \ulcorner A \urcorner & u \text{ fresh}
\end{array}$$

Please note that the translation changes α to α° in the translation of quantifiers (α° was defined in §3). Judgements are translated as follows: $\ulcorner \{A\}\,M\ :_m \{B\} \urcorner \overset{def}{=} M :_{\overline{u}} \overline{u}(m)\{\ulcorner A \urcorner\}\ulcorner B \urcorner$ (u fresh). This translation has the following properties.

Theorem 3. *1. The translation of judgements, when applied to rules, takes PCF-rules to derivable rules for PCF$^+$.*

2. $\vdash \{A\}\,M\ :_m \{B\}$ implies $\vdash \ulcorner \{A\}\,M\ :_m \{B\} \urcorner$, where derivability on the left is in the logic for PCF, on the right it's for PCF$^+$.

6 Conclusion

We have investigated program logics for a large class of stateless sequential control constructs. One construct not considered here are exceptions. Exceptions are a constrained form of jumping that is used to escape a context without the possibility of returning, a feature very useful for error handling. Exceptions are not included in the present logic because they are caught dynamically, which does not sit comfortably with our typing system. We believe that a simple extension of the logic presented here can easily account for exceptions. A second omission is that many programming languages with interesting control constructs also feature state. We believe that adding state to PCF$^+$or μPCF can be done easily with the help of content quantification [16].

Related Work. The present work builds upon a large body of preceding work on the semantics of control, including, but not limited to [11, 17, 21, 22, 25–28]. As mentioned, the investigation of logics for control manipulation was started by Clint and Hoare [10]. It has been revived by [2–4, 7, 24, 29, 32, 34] (the long version of the present paper will feature a more comprehensive discussion). None of these approaches investigates logics for fully-fledged higher-order control constructs like `callcc`.

The present work adds a new member to a family of logics for ML-like languages [5, 16, 18, 36], and integrates in a strong sense: e.g. all rules and axioms from [16] are, adapting the syntax, also valid for PCF$^+$and μPCF. We believe that all common CPS-transforms between PCF, PCF$^+$ and μPCF are logically fully abstract in the sense of [23]. This coherence between programming languages, their operational and axiomatic semantics, and compilations between each other paves the way for a comprehensive proof-compilation infrastructure for ML-like languages.

Rely/guarantee based reasoning was introduced in [20]. Internalising rely/guarantee reasoning into the program logic itself by way of rely/guarantee formulae was first proposed in [30, 31] and has been used in Ambient Logics [9] and in expressive typing systems [8]. The use of tensor is also found in [30, 31], and has been advocated by Winskel [35]. In all cases the context is concurrency, not sequential control.

A preliminary version of the present work was finished in 2007, and its key ideas, in particular rely/guarantee formulae and the tensor have since lead to a Hennessy-Milner logic for typed π-calculus [6]. Neither proof-rules nor axioms for higher-order control are investigated in [6]. Clarifying the relationship between the present logic and that of [6] is an interesting research question.

References

1. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Inf. & Comp. 163, 409–470 (2000)
2. Aspinall, D., Beringer, L., Hofmann, M., Loidl, H.-W., Momigliano, A.: A program logic for resource verification. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 34–49. Springer, Heidelberg (2004)
3. Bannwart, F., Müller, P.: A program logic for bytecode. ENTCS 141(1), 255–273 (2005)
4. Benton, N.: A Typed, Compositional Logic for a Stack-Based Abstract Machine. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 364–380. Springer, Heidelberg (2005)
5. Berger, M., Honda, K., Yoshida, N.: A logical analysis of aliasing for higher-order imperative functions. In: Proc. ICFP, pp. 280–293 (2005); Full version to appear in JFP
6. Berger, M., Honda, K., Yoshida, N.: Completeness and logical full abstraction in modal logics for typed mobile processes. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 99–111. Springer, Heidelberg (2008)
7. Beringer, L., Hofmann, M.: A bytecode logic for JML and types. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 389–405. Springer, Heidelberg (2006)
8. Caires, L.: Spatial-behavioral types, distributed services, and resources. In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661, pp. 98–115. Springer, Heidelberg (2007)
9. Cardelli, L., Gordon, A.D.: Anytime, Anywhere. Modal Logics for Mobile Ambients. In: Proc. POPL, pp. 365–377 (2000)
10. Clint, M., Hoare, C.A.R.: Program Proving: Jumps and Functions. Acta Informatica 1, 214–224 (1972)
11. Duba, B.F., Harper, R., MacQueen, D.: Typing First-Class Continuations in ML. In: Proc. POPL, pp. 163–173 (1991)
12. Harper, R., Lillibridge, M.: Operational Interpretations of an Extension of F_ω with Control Operators. Journal of Functional Programming 6(3), 393–417 (1996)
13. Honda, K.: Processes and games. ENTCS 71 (2002)
14. Honda, K., Berger, M., Yoshida, N.: Descriptive and Relative Completeness of Logics for Higher-Order Functions. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 360–371. Springer, Heidelberg (2006)
15. Honda, K., Yoshida, N.: A uniform type structure for secure information flow. In: POPL 2002, pp. 81–92. ACM Press, New York (2002); Full version to appear in ACM TOPLAS
16. Honda, K., Yoshida, N.: A compositional logic for polymorphic higher-order functions. In: Proc. PPDP 2004, pp. 191–202. ACM Press, New York (2004)
17. Honda, K., Yoshida, N., Berger, M.: Control in the π-calculus. In: Proc. CW 2004, ACM Press, New York (2004)
18. Honda, K., Yoshida, N., Berger, M.: An observationally complete program logic for imperative higher-order functions. In: LICS 2005, pp. 270–279 (2005)
19. Hyland, J.M.E., Ong, C.H.L.: On full abstraction for PCF. Inf. & Comp. 163, 285–408 (2000)
20. Jones, C.B.: Specification and Design of (Parallel) Programs. In: IFIP Congress, pp. 321–332 (1983)
21. Laird, J.: A Semantic Analysis of Control. PhD thesis, Univ. of Edinburgh (1998)

22. Longley, J.: When is a functional program not a functional program? SIGPLAN Not. 34(9), 1–7 (1999)
23. Longley, J., Plotkin, G.: Logical Full Abstraction and PCF. In: Tbilisi Symposium on Logic, Language and Information. CSLI (1998)
24. Ni, Z., Shao, Z.: Certified Assembly Programming with Embedded Code Pointers. In: Proc. POPL (2006)
25. Ong, C.-H.L., Stewart, C.A.: A Curry-Howard foundation for functional computation with control. In: Proc. POPL, pp. 215–227 (1997)
26. Parigot, M.: $\lambda\mu$-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In: Voronkov, A. (ed.) LPAR 1992. LNCS, vol. 624, pp. 190–201. Springer, Heidelberg (1992)
27. Plotkin, G.: Call-By-Name, Call-By-Value, and the λ-Calculus. TCS 1(2), 125–159 (1975)
28. Riecke, J.G., Thielecke, H.: Typed exceptions and continuations cannot macro-express each other. In: Wiedermann, J., Van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 635–644. Springer, Heidelberg (1999)
29. Saabas, A., Uustalu, T.: A Compositional Natural Semantics and Hoare Logic for Low-Level Languages. In: Proc. Workshop Structural Operational Semantics, SOS (2006)
30. Stirling, C.: A complete compositional proof system for a subset of CCS. In: Brauer, W. (ed.) ICALP 1985. LNCS, vol. 194, pp. 475–486. Springer, Heidelberg (1985)
31. Stirling, C.: Modal logics for communicating systems. TCS 49, 311–347 (1987)
32. Tan, G., Appel, A.W.: A Compositional Logic for Control Flow. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 80–94. Springer, Heidelberg (2005)
33. Thielecke, H.: Continuations, functions and jumps. Bulletin of EATCS, Logic Column 8 (1999)
34. Thielecke, H.: Frame rules from answer types for code pointers. In: Proc. POPL, pp. 309–319 (2006)
35. Winskel, G.: A complete proof system for SCCS with modal assertions. In: Maheshwari, S.N. (ed.) FSTTCS 1985. LNCS, vol. 206, pp. 392–410. Springer, Heidelberg (1985)
36. Yoshida, N., Honda, K., Berger, M.: Logical reasoning for higher-order functions with local state. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 361–377. Springer, Heidelberg (2007)

Modular Schedulability Analysis of Concurrent Objects in Creol[⋆]

Frank de Boer[1], Tom Chothia[1,2], and Mohammad Mahdi Jaghoori[1]

[1] CWI, Kruislaan 413, Amsterdam, The Netherlands
[2] School of Computer Science, University of Birmingham, UK
{jaghouri,t.chothia,f.s.de.boer}@cwi.nl

Abstract. We present an automata theoretic framework for modular schedulability analysis of real time asynchronous objects modeled in the language Creol. In previous work we analyzed the schedulability of objects modeled as Timed Automata. In this paper, we extend this framework to support enabling conditions for methods and replies to messages and we extend the Creol language to allow the specification of real time information. We provide an algorithm for automatically translating Creol code annotated with this real time information to timed automata. This translation handles synchronization mechanisms in Creol, as well as processor release points. With this translation algorithm, we can analyze end-to-end deadlines, i.e., the deadline on the time since a message is sent until a reply is received.

1 Introduction

Analyzing schedulability of a real time system consists of checking whether all tasks are accomplished within their deadlines. We employed automata theory in our previous work [7,8] to provide a high-level framework for modular schedulability analysis of asynchronous concurrent objects. Concurrent objects, having a dedicated processor each, are analyzed individually for schedulability with respect to their behavioral interfaces. A behavioral interface specifies at a high level and in the most general terms how an object may be used. As in modular verification [11], which is based on assume-guarantee reasoning, individually schedulable objects can be used in systems *compatible* with their behavioral interfaces. The schedulability of such systems is then guaranteed. Compatibility being subject to state space explosion can be efficiently tested [8]. Schedulability analysis and compatibility checking can be performed in Uppaal [12].

In this paper, we show the application of the modular schedulability analysis framework to Creol [9]. Creol is a full-fledged object-oriented modeling language based on concurrent objects. Creol objects communicate by asynchronous message passing, where receiving a message starts a new process in the object for executing the corresponding method. The processes in each object may be

[⋆] This work is partly funded by the European IST-33826 STREP project CREDO on Modeling and Analysis of Evolutionary Structures for Distributed Services.

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 212–227, 2010.

interleaved only at *processor release points*, i.e., the running process may be interrupted only when it voluntarily releases the processor. Then a process is nondeterministically taken out of the process queue and executed (called context-switch). In the case of a conditional release point, the running process is interleaved if and only if the condition is false. The rest of the method will be suspended until this condition is satisfied. Furthermore, an object can wait for a reply to a message it sends. This way one can model synchronous communication. If waiting is not performed in a release point, the waiting process will block the object (cf. Section 4).

Creol is a strongly typed modeling language supporting for instance multiple inheritance and type-safe dynamic class upgrades [15]. Schedulability analysis can be seen as a complementary feature for Creol. In this paper, we show how to add real time information to a Creol model. For every statement, the modeler can specify its (best- and worst-case) execution time. All method calls are given a deadline. In addition, an object is assigned a scheduling strategy (e.g., Earliest Deadline First, Fixed Priority Scheduling, etc.) to resolve the nondeterminism in selecting from among the enabled processes.

The object model of the schedulability analysis framework is similar to that of Creol, as objects have dedicated processors and communicate by asynchronous message passing. However, methods in this framework are modeled as timed automata [1] and run atomically to the end. The actions of these automata are sending messages. A self call, i.e., a message sent to the object itself, can be assigned an explicit deadline; otherwise, the called method (subtask) inherits the remaining deadline of the parent task (called delegation).

In this paper in Section 5, we provide an algorithm for extracting timed automata from Creol code. When the processor is released in the middle of a method, the rest of the method is modeled as a new automaton modeling the subtask. The subtask is added to the queue by means of a delegation self call before releasing the processor. Thus the subtask inherits the remaining deadline of the parent task. The subtask is, however, *disabled* as long as the condition of the release point is false. To model this, we need to extend the framework such that methods are assigned enabling conditions (cf. Section 3). The scheduler may schedule a task only if its enabling condition evaluates to true at the time of context switch. The subtasks generated by processing the release points are given as enabling condition the condition of the release point.

In Creol, in addition to messages, objects can send ‘replies’. Ideally each reply should correspond to a method invocation, but this leads to an infinite state model for a nonterminating system. Instead, method invocations are statically labeled and we associate replies to the labels. Thus, replies to different invocations associated with the same label are not distinguished. Replies should also be covered in the behavioral interface. Receiving a reply does not trigger a new task, but it may enable a suspended task that has been waiting for it. The scheduler automata model is adapted to handle replies properly. The compatibility check is also enhanced to include checking the correct and in time exchange of replies. The deadline until a reply is received is called an end-to-end deadline. Timely replies ensure end-to-end deadlines.

Related Work. Schedulability has been studied for actor languages [14] and event driven distributed systems [6]. Unlike these works, we work with non-uniformly recurring tasks as in task automata [5] which fits better the nature of message passing in object-oriented languages. The advantage of our work over task automata is that tasks are specified and may in turn create new tasks. Furthermore, a task may give up the processor before it is accomplished. Finally, we deal with end-to-end deadlines as a caller can wait for a reply from the callee.

The works of [2,10] is similar to ours as they extract automata from code for schedulability analysis. However, first of all they deal with programming languages and timings are usually obtained by profiling the real system. Our work is applied on a model before the implementation of the real system. Therefore, our main focus is on studying different scheduling policies and design decisions. More importantly, we address schedulability in a modular way.

A characteristic of our work is modularity. A behavioral interface models the most general message arrival pattern for an object. A behavioral interface can be viewed as a contract as in 'design by contract' [13] or as a most general assumption in modular model checking [11] (based on assume-guarantee reasoning); schedulability is guaranteed if the real use of the object satisfies this assumption. In the literature, a model of the environment is usually the task generation scheme in a specific situation. For example in TAXYS [2], different models of the environment can be used to check schedulability of the application in different situations. However, a behavioral interface in our analysis covers all allowable usages of the object, and is thus an over-approximation of all environments in which the object can be used. This adds to the modularity of our approach; every use of the object foreseen in the interface is verified to be schedulable.

2 Preliminaries

In this section, we define timed automata as it forms the basis of our analyses.

Definition 1 (Timed Automata). *Suppose $\mathcal{B}(C)$ is the set of all clock constraints on the set of clocks C. A timed automaton over actions Σ and clocks C is a tuple $\langle L, l_0, \longrightarrow, I\rangle$ representing*

- *a finite set of locations L (including an initial location l_0);*
- *the set of edges $\longrightarrow \subseteq L \times \mathcal{B}(C) \times \Sigma \times 2^C \times L$; and,*
- *a function $I : L \mapsto \mathcal{B}(C)$ assigning an invariant to each location.*

An edge (l, g, a, r, l') implies that action 'a' may change the location l to l' by resetting the clocks in r, if the clock constraints in g (as well as the invariant of l') hold. Since we use Uppaal [12], we allow defining variables of type boolean and bounded integers. Variables can appear in guards and updates.

A timed automaton is called *deterministic* if and only if for each $a \in Act$, if there are two edges (l, g, a, r, l') and (l, g', a, r', l'') from l labeled by the same action a then the guards g and g' are disjoint (i.e., $g \wedge g'$ is unsatisfiable).

Networks of timed automata. A system may be described as a collection of timed automata communicating with each other. In these automata, the action set is partitioned into input, output and internal actions. The behavior of the system is defined as the parallel composition of those automata $A_1 \parallel \cdots \parallel A_n$. Semantically, the system can delay if all automata can delay and can perform an action if one of the automata can perform an internal action or if two automata can synchronize on complementary actions (inputs and outputs are complementary). In a network of timed automata, variables can be defined locally for one automaton, globally (shared between all automata), or as parameters to the automata.

A location can be marked *urgent* in an automaton to indicate that the automaton cannot spend any time in that location. This is equivalent to resetting a fresh clock x in all of its incoming edges and adding an invariant $x \leq 0$ to the location. In a network of timed automata, the enabled transitions from an urgent location may be interleaved with the enabled transitions from other automata (while time is frozen). Like urgent locations, *committed* locations freeze time; furthermore, if any process is in a committed location, the next step must involve an edge from one of the committed locations.

3 The Modular Schedulability Analysis Framework

In this section, we present the automata-theoretic framework for modular schedulability analysis of asynchronous objects. The framework in [7,8] is extended here such that methods (and their corresponding messages) have enabling conditions. In addition, methods can send reply signals implying that the method execution has finished. This enables modeling Creol synchronization mechanisms.

Modeling behavioral interfaces. The abstract behavior of an object is specified in its behavioral interface. This interface consists of the messages the object may receive and send and provides an overview of the object behavior in a single automaton. It should also contain the reply signals the object may receive. A behavioral interface can also be seen as an abstraction (over-approximation) of the environments that can communicate with the object. A behavioral interface abstracts from specific method implementations, the queue in the object and the scheduling strategy.

We assume two finite global sets: $\mathcal{M}$ for method names and $\mathcal{T}$ for labels. Sending and receiving messages are written as $m!$ and $m?$, respectively. Sending a message can be labeled with $t \in \mathcal{T}$. Sending and receiving a reply associated to the label t are written as $t!$ and $t?$, respectively. A behavioral interface B providing a set of method names $M_B \subseteq \mathcal{M}$ is formally defined as a deterministic timed automaton over alphabet Act^B such that Act^B is partitioned into three sets of actions:

- object outputs received by the environment: $Act^B_O = \{m?|m \in \mathcal{M} \wedge m \notin M_B\}$
- object inputs sent by the environment: $Act^B_I = \{m(d)!|m \in M_B \wedge d \in \mathbb{N}\}$
- replies to the object outputs: $Act^B_r = \{t!|t \in \mathcal{T}\}$

The integer d associated to input actions represents a deadline. A correct implementation of the object should be able to finish method m before d time units. The methods M_B must exist in the classes implementing the interface B. Other methods are sent by the object and should be handled by the environment.

A behavioral interface includes the replies received by the object, because they are necessary for the schedulability of the methods waiting for the corresponding replies. These will also be used in compatibility checking to make sure that other objects provide timely reply signals.

Modeling classes. One can define a class as a set of methods implementing a specific behavioral interface. A class R implementing the behavioral interface B is a set $\{(m_1, E_1, A_1), \ldots, (m_n, E_n, A_n)\}$ of methods, where

– $M_R = \{m_1, \ldots, m_n\} \subseteq \mathcal{M}$ is a set of method names such that $M_B \subseteq M_R$;
– for all i, $1 \le i \le n$, A_i is a timed automaton representing method m_i with the alphabet $Act_i = \{m! | m \in M_R\} \cup \{m(d)! \mid m \in \mathcal{M} \wedge d \in \mathbb{N}\} \cup \{t? | t \in \mathcal{T}\}$;
– for all i, $1 \le i \le n$, E_i is the enabling condition for m_i.

Classes have an *initial* method which is implicitly called upon initialization and is used for the system startup. Method automata only send messages or wait for replies while computations are abstracted into time delays. Receiving messages (and buffering them) is handled by the scheduler automata explained next. Sending a message $m \in M_R$ is called a self call. Self calls may or may not be assigned an explicit deadline. The self calls with no explicit deadline are called *delegation*. Delegation implies that the internal task (triggered by the self call) is in fact the continuation of the parent task; therefore, the delegated task inherits the (remaining) deadline of the task that triggers it. As explained in the next section, delegation is essential in modeling Creol models correctly.

Modeling schedulers. A *scheduler automaton* implements a queue for storing messages and their deadlines. It is strongly input enabled, i.e., it can receive any message in M_R at any time. Whenever a method is finished, the scheduler selects another *enabled* message from the queue (based on its scheduling strategy) and starts the corresponding method (called context-switch). A message in the queue is enabled if its enabling condition evaluates to true. We have shown in [7] that we may put a finite bound on the queue length and still derive schedulability results that hold for any queue length (cf. next subsection). An Error location is reachable when a queue overflow occurs or a task misses its deadline.

Due to lack of space, for the details of modeling a scheduler in Uppaal and handling the deadlines using clocks we refer to our previous work [8]. We explain how to extend it here to support enabling conditions and replies. An enabling condition may include the availability of a reply. Since enabling conditions do not depend on clock values, and are statically defined for each method, we can define in Uppaal a boolean function to evaluate the enabling condition for each method when needed. An example of this function is given in Section 6. The selection strategy (which is specified as a guard) is then conjuncted with the result of the evaluation of this function.

As explained in Section 5, a reply is modeled by setting to true the variable associated to its corresponding label. However, when all the processes in the queue are disabled, receiving a reply may enable one of these processes. To handle this situation, we require the behavioral interface (in individual object analysis) or the replier object (when objects are put together) to synchronize with the scheduler on the reply channel. The scheduler then has the chance to select the enabled process for execution in the same way as in context-switch.

Modular Schedulability Analysis. An object is an instance of a class together with a scheduler automaton. To analyze an object in isolation, we need to restrict the possible ways in which the methods of this object could be called. Therefore, we only consider the incoming method calls specified in its behavioral interface. Receiving a message from another object (i.e., an input action in the behavioral interface) creates a new task (for handling that message) and adds it to the queue. The behavioral interface doesn't capture (internal tasks triggered by) self calls. In order to analyze the schedulability of an object, one needs to consider both the internal tasks and the tasks triggered by the (behavioral interface, which abstractly models the acceptable) environment.

We can generate the possible behaviors of an object by making a network of timed automata consisting of its method automata, behavioral interface automaton B and a concrete scheduler automaton. The inputs of B written as $m!$ will match with inputs in the scheduler written as $m?$ and the outputs of B written as $m?$ will match outputs of method automata written as $m!$.

An object is schedulable, i.e., all tasks finish within their deadlines, if and only if the scheduler cannot reach the Error location with a queue length of $\lceil d_{max}/b_{min} \rceil$, where d_{max} is the longest deadline for any method called on any transition of the automata (method automata or the input actions of the behavioral interface) and b_{min} is the shortest termination time of any of the method automata [7]. We can calculate the best case runtime for timed automata as shown by Courcoubetis and Yannakakis [3].

Once an object is verified to be schedulable with respect to its behavioral interface, it can be used as an off-the-shelf component. To ensure the schedulability of a system composed of individually schedulable objects, we need to make sure their real use is *compatible* with their expected use specified in the behavioral interfaces. The product of the behavioral interfaces, called B, shows the acceptable sequences of messages that may be communicated between the objects. Compatibility is defined as the inclusion of the visible traces of the system in the traces of B [8].

To avoid state-space explosion, we test compatibility. A trace is taken from B and turned into a test case by adding **Fail**, **Pass** and **Inconc** locations. Deviations from the trace either lead to **Inconc**, when the step is allowed in B, or otherwise lead to **Fail**. The submission of a test case consists of having it synchronize with the system. This makes the system take the steps specified in the original trace. The **Fail** location is reachable if and only if the system is incompatible with B along this trace. This property, called nonlaxness, as well as soundness, are proved in our previous work [8].

4 Real-Time Creol

Creol [9] is an object oriented modeling language for distributed systems. Creol fits our schedulability analysis framework, as a model consists of concurrent objects which communicate by asynchronous message passing. However, method definitions are more complex and may release the processor or wait for a reply to a message. In this section, we explain briefly the Creol modeling language and show how to add real time information to Creol code.

We abstract from method parameters and dynamic object creation. However, classes can have parameters. Class instances can communicate by objects given as class parameters, called the *known objects*. We can thus define the static topology of the system. The class behavior is defined in its methods, where a method is a sequence of statements separated by semicolon. A simplified syntax for Creol covered in our translation is given in Figure 1. For expressions, we assume the syntax that is accepted by UPPAAL.

Methods can have processor release points which define interleaving points explicitly. When a process is executing, it is not interrupted until it finishes or reaches a release point. Release points can be conditional, written as **await** g. If g is satisfied, the process keeps the control; otherwise, it releases the processor. When the processor is free, an enabled process is nondeterministically selected and started. The suspended process will be enabled when the guard evaluates to true. The **release** statement unconditionally releases the processor and the continuation of the process is immediately enabled.

If a method invocation p is associated with a label t, written as t!p(), the sender can wait for a reply using the blocking statement t? or in a nonblocking way by including t? in a release point, e.g., as in **await** t?. A reply is sent back automatically when the called method finishes. Before the reply is available, executing **await** t? releases the processor whereas the blocking statement t? does not. While the processor is not released, the other processes in the object do not get a chance for execution.

In standard Creol, different invocations of a method call are associated with different values of the label. For instance executing the statement t!p() twice results in two instances of the label t. Dynamic labels give rise to an infinite state space for non-terminating reactive systems. To be able to perform model checking, we treat every label as a static tag. Therefore, different invocations of

g	**: Guard**
b	**: Boolean**
t	**: Label**
m	**: Method**
p	**: Invocation**
x	**: Object**
s	**: Statement**
v	**: Variable**
e	**: Expression**
N	**: Identifier**

$$\begin{array}{l}
g \ ::= \ b \mid t? \mid !g \mid g \wedge g \\
p \ ::= \ x.m \mid m \\
S \ ::= \ s \mid s; S \\
s \ ::= \ v := e \mid !p() \mid t!p() \mid t? \mid \textbf{release} \mid \textbf{await}\ g \\
\qquad \mid \textbf{if}\ b\ \textbf{then}\ S\ \textbf{else}\ S\ \textbf{fi} \mid \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od} \\
Vdcl \ ::= \ N : [int \mid bool] \\
mtd \ ::= \ \textbf{op}\ N \ == \ S \\
cl \ ::= \ \textbf{class}\ N([Vdcl]^{*}_{,})\ \textbf{begin}\ [\textbf{var}\ Vdcl]^{*}\ [mtd]^{+}\ \textbf{end}
\end{array}$$

Fig. 1. The simplified grammar for the covered subset of Creol (adapted from [9])

```
class MutEx (left: Entity, right: Entity) begin
  var taken : bool
  op initial ==
     taken := false          /*@b1 @w1 : time delay*/
  op reqL ==
     await !taken;           /*@b1 @w2 : best and worst case*/
     taken := true;          /*@b1 @w1*/
     l ! left.grant();       /*@b4 @w4 @d10 : d = deadline*/
     await l?;               /*@b2 @w2*/
     taken := false          /*@b1 @w1*/
  op reqR ==
     await !taken ;          /*@b1 @w2*/
     taken := true;          /*@b1 @w1*/
     r ! right.grant();      /*@b4 @w4 @d10*/
     await r?;               /*@b2 @w2*/
     taken := false          /*@b1 @w1*/
end
```

Fig. 2. A Creol class for mutual exclusion with timing information

a method call with the same label are not distinguished in our framework. Alternatively, one could associate replies to message names, but this is too restrictive. By associating replies to labels, we can still distinguish the same message sent from different methods with different labels.

Adding Real-Time. The modeler should specify for every statement how long it takes to execute. The directives @b and @w are used for specifying the best-case and worst-case execution times for each statement. We assume some default execution time for different types of statements, e.g., for checking a guard, assignment or sending a message. The default value is used when no execution time is provided. Furthermore, every method call, including self calls, must be associated with a deadline using @d directive. This deadline specifies the relative time before which the corresponding method should be scheduled and executed. Since we do not have message transmission delays, the deadline expresses the time until a reply is received. Thus, it corresponds to an *end-to-end* deadline.

A worst-case execution time delay for a blocking statement t? is ignored. This statement may take so long as the deadline specified for the corresponding method call. In other words, we assume that external calls finish within their deadline. This is a fair assumption as long as individually schedulable objects are meant to be used in environments *compatible* with their behavioral interfaces.

Example 1. Figure 2 shows the Creol code for a mutual exclusion handler object annotated with timing information. An instance of MutEx should be provided with two instances of a class Entity representing the two objects (on its left and right) trying to get hold of the MutEx object. To do so, they may call reqL or reqR, respectively. The request is suspended if the object is already taken; otherwise, it is granted. The MutEx waits until the requester entity finishes its operation (in its grant method).

5 Generating Timed Automata from Creol

In this section, we explain the algorithm for automatically deriving automata from Creol code. We assume that the given Creol models are correctly typed and annotated with timing information. We use the same syntax for expressions and assignments in Creol, as is used by Uppaal. This allows for a more direct translation. For the sake of simplicity, we abstract from parameter passing, however, it can be modeled in Uppaal, by extending the queue to hold the parameters.

In applying the framework (cf. Section 3), the idea is that Creol classes are modeled as classes in the framework, and methods are represented by timed automata. In the next subsection, we explain how to extract timed automata from Creol code for methods. A class is then modeled by collecting the automata representing its methods. Every class should also be accompanied by a behavioral interface specification (using timed automata) and a scheduler automaton.

There are two major complications in this translation. Firstly, methods may release the processor before their completion. In these cases, the rest of the method is modeled as a sub-task in a separate automaton. Since the sub-task should inherit the deadline of the original task, it is put back into the queue using the *delegation* mechanism (cf. Section 3). The condition of the release point is used as its enabling condition.

When the condition in a release point includes t?, i.e., waiting until the reply to the call with label t is available, we replace t with a global variable which is set to true by the callee when it finishes. The behavioral interface should capture the expected time when the reply is made ready. This time must match the deadline provided when performing the corresponding call. See Section 6 for an example.

The second complication is how we can map a possibly infinite state Creol model to finite state automata. We do this by abstracting away some information: variables from a finite domain can be mapped to themselves but conditions on potentially infinite variables are mapped to true, we perform this mapping with the function f in Figure 3. The user must state which variables must be mapped to true. This means that our automata over-approximate the behaviors of the Creol model. A more advanced abstraction would map potentially infinite variables to finite domains in order to narrow the over-approximation. Due to lack of space we do not elaborate on this abstraction in this paper.

5.1 The Translation Algorithm

In this section, we present our translation algorithm, which can be seen as a custom form of graph transformations [4]. For each method, we start by an automaton with two locations and one transition where the final location is urgent

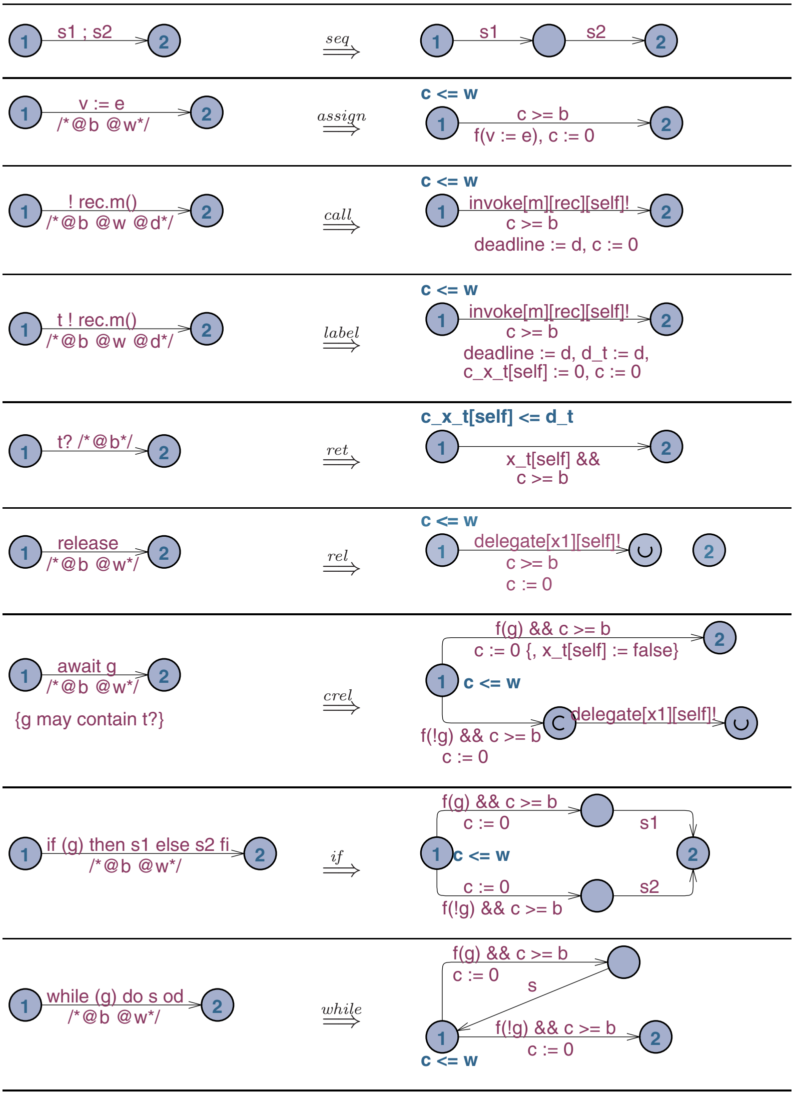


Fig. 3. Automata expansion rules

(marked U). We put the whole body of the method as a big statement on this transition. This automaton (treated as a graph) is expanded by matching and expanding the statements as in Figure 3. The expansion and finalization of this automaton is explained in the sequel. As part of the finalization, the automaton is duplicated for each possible release point, and each of these automata is given a proper enabling condition.

Expanding Creol statements. The expansion of statements works by applying repeatedly the rules in Figure 3. The labels 1 and 2 on the locations show the correspondence between the left and right hand sides of the rules. The locations marked U all correspond to the final location of the starting automaton (see above). These rules are applied as long as they are enabled, i.e., the label on left-hand side can be matched with a transition (according to the grammar in Section 4). The rule is then applied by removing this transition, and adding instead the new transitions and locations on the right hand side between the locations marked 1 and 2. It can be easily shown that the order of applying the rules is not important [4, Section 3.3]. When applying these rules, the following notes should be considered.

Since different methods in a class are modeled in different automata, the class variables need to be defined globally. Variables are defined as arrays, and the identity of the object is used to distinguish between variables of different instances of the same class. The function f is used to add [self] to every variable used in an expression (as in the *assign* rule). For every label t, a boolean variable x_t, a clock c_x_t and an integer d_t are defined such that these names are unique. Similar to variables, x_t and c_x_t are also treated as arrays indexed by *self*. The value of d_t is the same for all objects of this type.

The rules *call* and *label* translates the calls. If a label is not present, only the deadline is set to d. The calls are translated into a synchronization on the "invoke" channel. This implies that we do not allow explicit delegation. Instead, release points should be used as a high-level construct for breaking a method into subtasks. Every method should be self-contained and correspond to a single job, which is given a static deadline.

Every time the rules *rel* and *crel* are applied, the text $x1$ should be replaced by a fresh name, not used already in the model. This name will be used as the name of the sub-task modeling the continuation of the method. A conditional release point may also include a check whether the response to a previous method call labeled by 't' is available. In this case, a fresh boolean variable (written x_t, e.g., in the *ret* rule) is introduced which will be set to true by the callee (or in the behavioral interface, when performing individual schedulability analysis). In this case, a check on 't?' can be replaced by a check on x_t. The variable x_t must be reset after it is used for enabling a suspended method. This is done at the transition labeled g on the right side of the rule *crel* (shown in curly braces).

The blocking statement t? is translated to a transition with a guard waiting until x_t is set to true (*ret* rule). This transition takes a maximum time equal to the deadline of the corresponding method call. This is achieved by resetting a clock c_x_t and setting d_t to the corresponding deadline value at the *label* rule, and checking c_x_t against d_t at the *ret* rule.

The execution times for every statement are applied with the help of a clock c. In the case of *if*, *while* and *crel*, it is treated as the delay before evaluating the guard. The delegation step in a conditional release is performed in zero time (using a committed location marked C). The time delays provided by the modeler can in principle model the computation that is abstracted away when modeling the system at a high level.

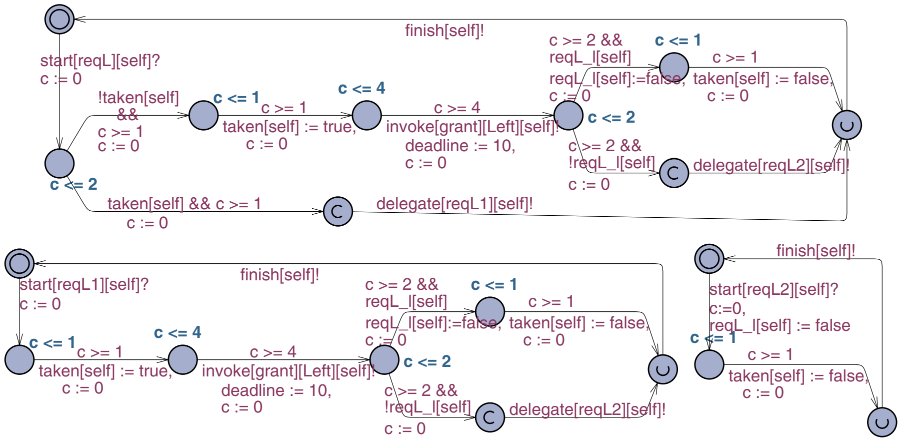


Fig. 4. Three automata are generated for the method reqL in Figure 2. The unreachable locations in reqL1 and reqL2 are not shown. The enabling conditions for reqL, reqL1 and reqL2 are true, !taken and reqL_l, respectively.

Finalizing the methods. The rules given above generate an automaton with only UPPAAL primitive actions on its labels, i.e., assignments, channel synchronizations or guards. This automaton called m (where m is the name of the method) has one *start* location with no incoming transitions and one urgent *final* locations marked 'U'. This automaton will schedule a sub-task whenever the method should release the processor. This sub-task, when scheduled, should continue where the parent task left off.

To complete the modeling of release points, the automaton m is duplicated once for each release point to model the corresponding subtask. For the duplicated automata, each automaton is given the name generated when processing the corresponding release point (the new name $x1$ above). The start location of each of these automata is changed to the location where the resumed sub-task should start; this corresponds to the location marked 2 in the *rel* and *crel* rules.

We finalize each automaton, n, by adding the following:

- a new location marked *initial*,
- a transition with a synchronization on "start [n][self]?" from the initial location to the start location (defined above) in order to enable the scheduler to start this task. This transition must have an update "c := 0".
- a transition with the label "finish [self]!" from the urgent final location to the initial location. This allows restarting the method if it is called again.

The guards of conditional release points are set as the enabling conditions of the corresponding automata. Other automata have *true* as their enabling condition. If the enabling condition of a method automaton requires waiting for a reply associated to a label t, the variable x_t needs to be reset on the 'start' transition. We illustrate this process in the next section.

5.2 End-to-End Deadlines

When calling a method with a deadline and waiting for the response later in the method, we are, in fact, enforcing an end-to-end deadline on the method call. This deadline must be enforced on the behavioral interface for the arrival of the response (with the assumption that only individually schedulable objects will be used together). This is crucial to the schedulability of the caller object. If no such restriction on the arrival of the response is enforced, the caller may be provided with the reply too late, therefore it will miss the deadline for performing the task enforced by the method that is waiting for the reply. This restriction is reflected in the extended compatibility check (cf. Section 6.1).

6 Schedulability Analysis of Creol

Having generated method automata for a Creol class, we can perform schedulability analysis as explained in Section 3. Figure 4 shows the automata generated for the method reqL from Figure 2. The automata for reqR are similar. Since reqL has two release points, it results in three method automata; one for the method itself, and one for each possible subtask generated by releasing the processor. Figure 5 shows the function modeling the enabling conditions for MutEx.

To be able to perform schedulability analysis for this object, we need the behavioral interface specification, as in Figure 6. When put together with its behavioral interface, the object will be checked to make sure it finishes the incoming messages (‘requests’ in MutEx example) within their deadlines. The outputs of the object are also ensured to conform to the behavioral interface. For each output, the object may wait for a reply. The behavioral interface provides the expected replies assuming that the end-to-end deadlines for the output messages are satisfied. The object is analyzed to find the proper scheduling strategy. For schedulability of MutEx, an ‘earliest deadline first’ strategy is needed to favor old requests to new ones.

```
bool isEnabled (int msg, int self) {
    if (msg == reqL1) return !taken[self];
    else if (msg == reqL2) return reqL_l[self];
    else if (msg == reqR1) return !taken[self];
    else if (msg == reqR2) return reqR_r[self];
    return true; // other method automata
}
```

Fig. 5. An example function capturing enabling conditions

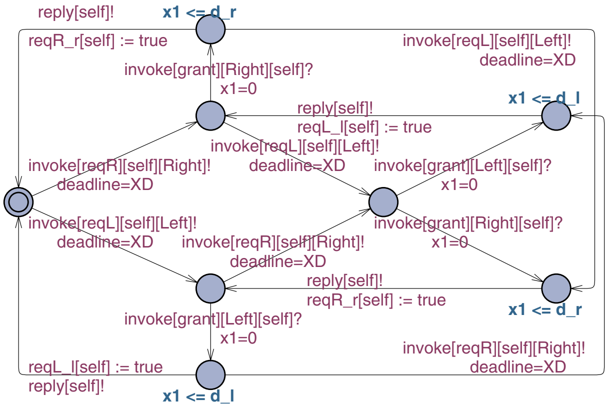

reply[self]!
x1 <= d_r
reqR_r[self] := true
invoke[reqL][self][Left]!
deadline=XD
invoke[grant][Right][self]?
x1=0
x1 <= d_l
reply[self]!
reqL_l[self] := true
invoke[reqL][self][Left]!
invoke[reqR][self][Right]!
deadline=XD
deadline=XD
invoke[grant][Left][self]?
x1=0
invoke[reqL][self][Left]!
deadline=XD
invoke[reqR][self][Right]!
deadline=XD
invoke[grant][Right][self]?
x1=0
reply[self]!
reqR_r[self] := true
x1 <= d_r
invoke[grant][Left][self]?
x1=0
invoke[reqR][self][Right]!
deadline=XD
reqL_l[self] := true
reply[self]!
x1 <= d_l

Fig. 6. The behavioral interface for the MutEx object

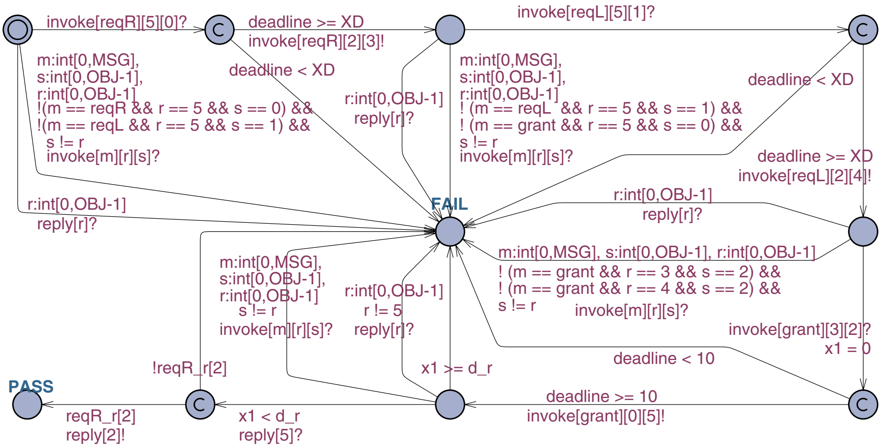


Fig. 7. Test case for compatibility including reply to messages

6.1 Checking Compatibility

Once schedulability is established, the objects can be put together to form a system. Then we should check if the usage of the objects in the system is compatible with their behavioral interfaces.

A complication in forming a system is making methods send reply signals. If when calling a method, a label t is assigned to the method call, the called method should set the variable corresponding to this label, namely x_t (cf. previous section), to true and send a reply signal by synchronizing on the reply channel when the method finishes. In individual object analysis, this is performed by the behavioral interface. When making a system, this can be added automatically by a static check of the model.

When testing for compatibility, we should also check if the correct replies to methods are received. To do so, the objects need to synchronize with the test case on reply channel (as well as invoke and delegate channels). In the case of sending messages, the test-case checks the compatibility of the deadline values. For replies, we check if the correct variable (corresponding to the correct label) is set to true. Compatibility thus ensures that replies arrive in time with respect to the end-to-end deadline requirements (cf. Section 5.2).

Figure 7 shows a test case for checking the compatibility of a system using the MutEx object. In this system, the deadline for grant method is assumed to be 10 in the behavioral interface of the Entity class. The test case does not show the **Inconc** location. Any message or reply not foreseen in the behavioral interfaces will lead to **Fail**.

7 Conclusions and Future Work

Creol is full-fledged object oriented modeling language with a formal semantics supporting strong typing. Schedulability analysis is added to Creol as a complementary feature in this paper. We investigated deriving automata from Creol code augmented with real time information. We can then apply to Creol the automata-theoretic framework for schedulability analysis. To this end, the framework is extended to support enabling conditions for methods. We adapted the compatibility check such that it includes checking for timely replies to the external method calls.

With this translation, we provide a solution to the schedulability analysis of concurrent object models without restricting the task generation to periodic patterns or pessimistic approximations. Instead, automata are used to enable modeling nonuniformly recurring tasks. We explained how ‘processor release points’ and synchronization mechanisms (waiting for a reply to an asynchronous call) are handled.

Due to lack of space, a formal proof of correctness was not given in this paper. This can be given based on the formal semantics of Creol and Timed automata. The Creol semantics is un-timed, therefore we either have to extend its semantics with real time, or we would have to ignore the timed aspects in our proof. We can then show an operational correspondence or bisimulation between the transition systems defined for a Creol model by its semantics, and the network of timed automata that the Creol model is translated to by the timed automata semantics.

As further work, one can study adding scheduling policies to the original semantics of Creol. This provides an execution platform for the schedulable Creol objects. We are currently working on an extension of the framework such that it supports multi-core processors, where each object can have more than one thread of execution. This enables us to analyze more realistic models.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Closse, E., Poize, M., Pulou, J., Sifakis, J., Venter, P., Weil, D., Yovine, S.: TAXYS: A tool for the development and verification of real-time embedded systems. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 391–395. Springer, Heidelberg (2001)
3. Courcoubetis, C., Yannakakis, M.: Minimum and maximum delay problems in real-time systems. Formal Methods in System Design 1(4), 385–415 (1992)
4. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theoretical Computer Science. Springer, Heidelberg (2006)
5. Fersman, E., Krcal, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Information and Computation 205(8), 1149–1172 (2007)
6. Garcia, J.J.G., Gutierrez, J.C.P., Harbour, M.G.: Schedulability analysis of distributed hard real-time systems with multiple-event synchronization. In: Proc. 12th Euromicro Conference on Real-Time Systems, pp. 15–24. IEEE, Los Alamitos (2000)
7. Jaghoori, M.M., de Boer, F.S., Chothia, T., Sirjani, M.: Schedulability of asynchronous real-time concurrent objects. J. Logic and Alg. Prog. 78(5), 402–416 (2009)
8. Jaghoori, M.M., Longuet, D., de Boer, F.S., Chothia, T.: Schedulability and compatibility of real time asynchronous objects. In: Proc. Real Time Systems Symposium, pp. 70–79. IEEE CS, Los Alamitos (2008)
9. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling 6(1), 35–58 (2007)
10. Kloukinas, C., Yovine, S.: Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems. In: Proc. 15th Euromicro Conference on Real-Time Systems (ECRTS 2003), pp. 287–294. IEEE Computer Society, Los Alamitos (2003)
11. Kupferman, O., Vardi, M.Y., Wolper, P.: Module checking. Information and Computation 164(2), 322–344 (2001)
12. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. STTT 1(1-2), 134–152 (1997)
13. Meyer, B.: Eiffel: The language. Prentice-Hall, Englewood Cliffs (1992);(first printing: 1991)
14. Nigro, L., Pupo, F.: Schedulability analysis of real time actor systems using coloured petri nets. In: Agha, G.A., De Cindio, F., Rozenberg, G. (eds.) APN 2001. LNCS, vol. 2001, pp. 493–513. Springer, Heidelberg (2001)
15. Yu, I.C., Johnsen, E.B., Owe, O.: Type-safe runtime class upgrades in Creol. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 202–217. Springer, Heidelberg (2006)

A Timed Calculus for Wireless Systems[*]

Massimo Merro and Eleonora Sibilio

Dipartimento di Informatica, Università degli Studi di Verona, Italy

Abstract. We propose a *timed process calculus* for *wireless systems*, paying attention in modelling *communication collisions*. The operational semantics of our calculus is given in terms of a *labelled transition system*. The calculus enjoys a number of desirable time properties such as (i) *time determinism*: the passage of time is deterministic; (ii) *patience*: devices will wait indefinitely until they can communicate; (iii) *maximal progress*: data transmissions cannot be delayed, they must occur as soon as a possibility for communication arises.

The main behavioural equality of our calculus is a timed variant of *barbed congruence*, a standard branching-time and contextually-defined program equivalence. As an efficient proof method for timed barbed congruence we define a labelled bisimilarity. We then apply our bisimulation proof-technique to prove a number of algebraic properties.

1 Introduction

Wireless technology spans from user applications such as personal area networks, ambient intelligence, and wireless local area networks, to real-time applications, such as cellular, and ad hoc networks. The IEEE 802.11 standard [1] contains a series of specifications for wireless LAN technologies. The basic building block of an 802.11 network is the *basic service set* (BBS), which is a set of stations that have successfully synchronised and that communicate using radio transceivers. In *independent* BBS (IBBS) stations communicate directly with each other without using any *distribution system*. IBBS networks are sometimes referred to as *ad hoc networks*. In this paper, we propose a formal model for IBBS networks paying particular attention to *communication interferences*. Communication interferences represent the main concern when evaluating the *network throughput*, i.e. the average rate of successful message delivery over a communication channel.

In concurrent systems, an interference occurs when the activity of a component is damaged or corrupted because of the activities of another component. In Ethernet-like networks communication channels are full-duplex; that is, a node can transmit and receive at the same time. As a consequence, *collisions* caused by two simultaneous transmissions are immediately detected and repaired by retransmitting the message after a randomly-chosen period of time. This is not possible in wireless networks where radio signals span over a limited area, called *transmission cell*, and channels are *half-duplex*: on a given channel, a device can

[*] This work was partially supported by the PRIN 2007 project "SOFT".

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 228–243, 2010.

either transmit or receive, but cannot do both at the same time. As a consequence, in wireless systems communication collisions can only be detected at destination by receivers exposed to different transmissions.

In the last twenty-five years, process calculi [2,3,4,5,6] have been intensively used to study the semantics of concurrent/distributed systems, and to develop verification techniques for such systems. In the literature, there exist a number of process calculi modelling wireless systems [7,8,9,10,11,12,13,14]. All these calculi rely on the presence of some MAC level protocol to avoid interferences. However, in wireless systems *collisions cannot be eliminated* although there exist protocols to reduce their occurrences (see, for instance, the IEEE 802.11 CSMA/CA protocol [1] for unicast communications). Due to their influence on networks' performance, communication collisions represent a serious problem that should be taken into account when modelling wireless systems.

Many protocols for wireless networks rely on a *common notion of time* among the devices, provided by some *clock synchronisation protocol.* Most clock synchronisation protocols for ad hoc networks [15,16,17,18,19,20] follow the "clock correction" approach correcting the local clock of each node to run in par with a global time scale.[1] This approach heavily rely on *network connectivity*. In a connected network all nodes are in touch with each other, although not always directly. Wireless networks are usually assumed to be connected; disconnected devices can be considered as not being part of the network as, in general, they need to re-authenticate to rejoin the network.

In this paper, we propose a *timed calculus* for wireless networks, called TCWS, in which all wireless devices are *synchronised* using a clock-correction synchronisation protocol. Our notion of time is modelled through a simple action σ, in the style of Hennessy and Regan [22], to denote idling until the next clock cycle. Time proceeds in *discrete* steps represented by occurrences of the action σ.

As in Hennessy and Regan's timed CCS [22], and Prasad's timed CBS [23], our TCWS enjoys three basic time properties:

- *time determinism*: the passage of time is deterministic, i.e. a node (and hence a network) can reach at most one new state by performing the action σ;
- *patience*: nodes will wait indefinitely until they can communicate;
- *maximal progress*: data transmissions cannot be delayed, they must occur as soon as a possibility for communication arises.

The operational semantics of our calculus is given in terms of a *labelled transition system* (LTS). As usual for ad hoc networks, the communication mechanism is *broadcast.* We provide a notion of network well-formedness to take into account node-uniqueness, network connectivity, transmission exposure, and transmission consistency. We then prove that our labelled transition semantics preserves network well-formedness.

A central concern in process calculi is to establish when two terms have the same *observable behaviour*, that it, they are indistinguishable in any context. *Behavioural equivalences* are fundamental for justifying program transformations.

[1] An excellent survey of existing clock synchronisation protocols for sensor networks (and more generally for ad-hoc networks) can be found in [21].

Table 1. The Syntax

Values			
u	$::=$	x	variable
	$\mid$	v	closed value
Networks:			
M, N	$::=$	$\mathbf{0}$	empty network
	$\mid$	$M_1 \mid M_2$	parallel composition
	$\mid$	$n[W]_t^{\nu}$	node
Processes:			
W	$::=$	P	inactive process
	$\mid$	A	active process
P, Q	$::=$	nil	termination
	$\mid$	$!\langle u \rangle.P$	broadcast
	$\mid$	$?(x).P$	receiver
	$\mid$	$\sigma.P$	delay
	$\mid$	$\lfloor ?(x).P \rfloor Q$	receiver with timeout
	$\mid$	$[u_1 = u_2]P, Q$	matching
	$\mid$	$H\langle \tilde{u} \rangle$	recursion
A	$::=$	$\langle v \rangle^t.P$	active sender
	$\mid$	$(x)_v.P$	active receiver

Our program equivalence is a timed variant of (weak) *reduction barbed congruence*, a branching-time contextually-defined program equivalence. Barbed equivalences [24] are simple and intuitive but difficult to use due to the quantification on all contexts. Simpler proof techniques are based on *labelled bisimilarities* [2], which are co-inductive relations that characterise the behaviour of processes using a *labelled transition system*. We define a *labelled bisimilarity* which is a proof method for timed reduction barbed congruence. We then apply our bisimulation proof-technique to prove a number of algebraic laws.

2 The Calculus

In Table 1, we define the syntax of TCWS in a two-level structure, a lower one for *processes* and a upper one for *networks*. We use letters $a, b, c, \ldots$ for logical names, x, y, z for *variables*, u for *values*, and v and w for *closed values*, i.e. values that do not contain free variables. Closed values actually denotes messages that are transmitted as TCP/IP packets. Packets contains a number of auxiliary informations such as the network address of the transmitter. So, sometimes we write $m{:}v$ to mean a message v transmitted by node m. With an abuse of notation, structured messages of the form $m{:}v$ are ranged by the same letters v and w. We write $\tilde{u}$ to denote a tuple $u_1, \ldots, u_k$ of values.

Networks are collections of nodes (which represent devices) running in parallel and using a unique common channel to communicate with each other. We use the symbol $\mathbf{0}$ to denote an empty network, while $M_1 \mid M_2$ represents the parallel composition of two sub-networks M_1 and M_2. Nodes cannot be created or destroyed. All nodes have the same transmission range. We write $n[P]^{\nu}_{t}$ for a node named n (the device network address) executing the sequential process P. The tag ν denotes the set of (the names of) the neighbours of n. Said in other words, ν contains all nodes in the transmission cell of n. In this manner, we model the network topology. Notice that the network topology could have been represented using some kind of restriction operator à la CCS over node names. We preferred our notation to keep at hand the neighbours of a node. The variable t is a semantic tag ranging over positive integers to represent *node exposure*. Thus, a node $n[W]^{\nu}_{t}$, with $t > 0$, is exposed to a transmission (or more transmissions) for the next t instants of time.

Processes are sequential and live within the nodes. For convenience, we distinguish between *non-active* and *active processes*. An active process is a process which is currently transmitting or receiving. An *active node* is a node with an active process inside. The symbol nil denotes the skip process. The sender process $!\langle v\rangle.P$ allows to broadcast the value v. Once the transmission starts the process evolves into the active sender process $\langle v\rangle^{\delta_v}.P$ which transmits the message v for the next δ_v time units, the time necessary to transmit v. In the construct $\langle v\rangle^{t}.P$ we require $t > 0$. The receiver process $?(x).P$ listens on the channel for incoming messages. Once the reception starts the process evolves into the active receiver process $(x)_w.P$ and starts receiving. When the channel becomes free the receiver calculates the CRC to check the integrity of the received packets. Upon successful reception the variable x of P is instantiated with the transmitted message w. The process $\sigma.P$ models sleeping for one time unit. The process $\lfloor ?(x).P\rfloor Q$ denotes a receiver with timeout. Intuitively, this process either starts receiving a value in the current instant of time, evolving into an active receiver, or it idles for one time unit, and then continues as Q. Process $[v_1 = v_2]P, Q$ is the standard "if then else" construct: it behaves as P if $v_1 = v_2$, and as Q otherwise. In processes $\sigma.P, ?(x).P$, $\lfloor ?(x).P\rfloor Q$, and $!\langle v\rangle.P$ the occurrence of process P is said to be *guarded*. We write $H\langle\tilde{v}\rangle$ to denote a process defined via a definition $H(\tilde{x}) \overset{\text{def}}{=} P$, with $\mid \tilde{x} \mid=\mid \tilde{v} \mid$, where $\tilde{x}$ contains all variables that appear free in P. Defining equations provide *guarded recursion*, since P may contain only guarded occurrences of process identifiers, such as H itself. We use a number of notational conventions. $\prod_{i\in I} M_i$ means the parallel composition of all sub-networks M_i, for $i \in I$. We write $!\langle v\rangle$ for $!\langle v\rangle.\mathsf{nil}$, and $\langle v\rangle^{\delta}$ for $\langle v\rangle^{\delta}.\mathsf{nil}$. We recall that in the active sender process $\langle v\rangle^{r}.P$ it holds that $r > 0$. However, sometimes, for convenience, we write $\langle v\rangle^{0}.P$ assuming the following syntactic equality $\langle v\rangle^{0}.P = P$. Similarly, an active receiver node $m[(x)_w.P]^{\nu}_{t}$ makes only sense if $t > 0$. However, for convenience, we sometimes write $m[(x)_w.P]^{\nu}_{0}$, which is syntactically equals to $m[\{{}^{w}\!/\!{}_{x}\}P]^{\nu}_{0}$.

In the terms $?(x).P$, $\lfloor ?(x).P\rfloor Q$, and $(x)_v.P$ the variable x is bound in P. This gives rise to the standard notion of α-conversion. We identify processes

Table 2. LTS - Process transitions

$$\text{(SndP)}\ \frac{-}{!\langle v\rangle.P \xrightarrow{\;!v\;} \langle v\rangle^{\delta_v}.P} \qquad \text{(RcvP)}\ \frac{-}{?(x).P \xrightarrow{\;?m:v\;} (x)_{m:v}.P}$$

$$\text{(RcvTO)}\ \frac{-}{\lfloor ?(x).P\rfloor Q \xrightarrow{\;?m:v\;} (x)_{m:v}.P} \qquad \text{(Timeout)}\ \frac{-}{\lfloor ?(x).P\rfloor Q \xrightarrow{\;\sigma\;} Q}$$

$$\text{(Nil-}\sigma\text{)}\ \frac{-}{\mathsf{nil} \xrightarrow{\;\sigma\;} \mathsf{nil}} \qquad \text{(Rcv-}\sigma\text{)}\ \frac{-}{?(x).P \xrightarrow{\;\sigma\;} ?(x).P}$$

$$\text{(Sigma)}\ \frac{-}{\sigma.P \xrightarrow{\;\sigma\;} P}$$

$$\text{(ActSnd)}\ \frac{r>0}{\langle v\rangle^{r}.P \xrightarrow{\;\sigma\;} \langle v\rangle^{r-1}.P} \qquad \text{(ActRcv)}\ \frac{-}{(x)_v.P \xrightarrow{\;\sigma\;} (x)_v.P}$$

and networks up to α-conversion. We assume there are no free variables in our networks. The absence of free variables in networks is trivially maintained as the network evolves. We write $\{^{v}\!/_{x}\}P$ for the substitution of the variable x with the value v in P. We define *structural congruence*, written $\equiv$, as the smallest congruence which is a commutative monoid with respect to the parallel operator.

Given a network M, $\mathsf{nds}(M)$ returns the names of the nodes which constitute the network M. For any network M, $\mathsf{actsnd}(M)$ and $\mathsf{actrcv}(M)$ return the set of active senders and active receivers of M, respectively. Thus, for instance, for $N = m[!\langle w\rangle]^{\nu}_{t} \mid n[\langle v\rangle^{r}.P]^{\nu'}_{t'}$ we have $\mathsf{nds}(N) = \{m,n\}$ and $\mathsf{actsnd}(N) = \{n\}$. Given a network M and an active sender $n \in \mathsf{actsnd}(M)$, the function $\mathsf{active}(n,M)$ says for how long the node will be transmitting. For instance, if N is the network defined as before, $\mathsf{active}(n,N) = r$. If n is not an active sender then $\mathsf{active}(n,N) = 0$. Finally, given a network M and a node $m \in \mathsf{nds}(M)$, the function $\mathsf{ngh}(m,M)$ returns the set of neighbours of m in M. Thus, for N defined as above $\mathsf{ngh}(m,N) = \nu$.

2.1 The Operational Semantics

We give the operational semantics of our calculus in terms of a Labelled Transition System (LTS). Table 2 contains a simple LTS for processes. Rules (SndP) and (RcvP) model the beginning of a transmission. In rule (SndP) a sender evolves into an active sender. For convention we assume that the transmission of a value v takes δ_v time units. In rule (RcvP) a receiver evolves into an active receiver $(x)_{m:v}.P$ where m is the transmitter's name and v is the value that is supposed to be received after δ_v instants of time. The process $\lfloor ?(x).P\rfloor Q$ can start a reception in the current instant of time, as $?(x).P$, or it can idle for one time unit evolving into Q. Rules (RcvTO) and (Timeout) model these two different

Table 3. LTS - Begin transmission

$$\text{(Snd)}\ \frac{P \xrightarrow{!v} A}{m[P]^{\nu}_{t} \xrightarrow{m!v} m[A]^{\nu}_{t}} \qquad \text{(Rcv)}\ \frac{m \in \nu \quad P \xrightarrow{?m:v} A}{n[P]^{\nu}_{0} \xrightarrow{m?v} n[A]^{\nu}_{\delta_v}}$$

$$\text{(RcvPar)}\ \frac{M \xrightarrow{m?v} M' \quad N \xrightarrow{m?v} N'}{M \mid N \xrightarrow{m?v} M' \mid N'} \qquad \text{(Sync)}\ \frac{M \xrightarrow{m!v} M' \quad N \xrightarrow{m?v} N'}{M \mid N \xrightarrow{m!v} M' \mid N'}$$

$$\text{(Coll)}\ \frac{m \in \nu \quad t'{:=}\mathsf{max}(t,\delta_v)}{n[(x)_w.P]^{\nu}_{t} \xrightarrow{m?v} n[(x)_{\perp}.P]^{\nu}_{t'}} \qquad \text{(Exp)}\ \frac{m \in \nu \quad W{\neq}(x)_w.P \quad t'{:=}\mathsf{max}(t,\delta_v)}{n[W]^{\nu}_{t} \xrightarrow{m?v} n[W]^{\nu}_{t'}}$$

$$\text{(OutRng)}\ \frac{m \notin \nu \quad m \neq n}{n[W]^{\nu}_{t} \xrightarrow{m?v} n[W]^{\nu}_{t}} \qquad \text{(Zero)}\ \frac{-}{\mathbf{0} \xrightarrow{m?v} \mathbf{0}}$$

behaviours, respectively. The remaining rules regards time passing. Rules (Nil-σ), (Rcv-σ), and (Sigma) are straightforward. In rule (ActSnd) the time necessary to conclude the transmission is decreased. In rule (ActRcv) the derivative does not chance as a reception terminates only when the channel is sensed free. Notice that sender processes do not perform σ-actions. This is to model the maximal progress property.

We have divided the LTS for networks in two sets of rules corresponding to the two main aspects of a wireless transmission. Table 3 contains the rules to model the initial synchronisation between the sender and its neighbours. Table 4 contains the rules for modelling time passing and transmission ending. Let us comment on the rules of Table 3. Rule (Snd) models a node starting a broadcast of message v to its neighbours in ν. By maximal progress, a node which is ready to transmit will not be delayed. Rule (Rcv) models the beginning of the reception of a message v transmitted by a station m. This happens only when the receiver is not exposed to other transmissions i.e. when the exposure indicator is equal to zero. The exposure indicator is then updated because node n will be exposed for the next δ_v instants of time. Rule (RcvPar) models multiple receptions. Rule (Sync) serves to synchronise the components of a network with a broadcast transmission originating from a node m. In rule (Coll) an active receiver n is exposed to a transmission originating from a node m. This transmission gives rise to a *collision* at n. Rule (Exp) models the exposure of a node n (which is not an active receiver) to a transmission originating from a transmitter m. In this case, n does not take part to the transmission. Notice that a node $n[?(x).P]^{\nu}_{0}$ might execute rule (Exp) instead of (Rcv). This is because a potential (synchronised) receiver might miss the synchronisation with the sender for several reasons (internal misbehaving, radio signals problems, etc). Such a situation will give rise to a failure at n (see rule (RcvFail) in Table 4). Rule (OutRng) regards nodes which are out of the range of a transmission originating from a node m. Rule

Table 4. LTS - Time passing/End transmission

$$\text{(Time-0)}\ \frac{W \xrightarrow{\sigma} W'}{n[W]^{\nu}_{0} \xrightarrow{\sigma} n[W']^{\nu}_{0}}$$

$$\text{(Time-t)}\ \frac{t>0 \quad W \xrightarrow{\sigma} W' \quad W \not\xrightarrow{?v}}{n[W]^{\nu}_{t} \xrightarrow{\sigma} n[W']^{\nu}_{t-1}} \qquad \text{(RcvFail)}\ \frac{t>0 \quad P \xrightarrow{?\bot} A}{n[P]^{\nu}_{t} \xrightarrow{\sigma} n[A]^{\nu}_{t-1}}$$

$$\text{(Zero-}\sigma\text{)}\ \frac{-}{\mathbf{0} \xrightarrow{\sigma} \mathbf{0}} \qquad \text{(Par-}\sigma\text{)}\ \frac{M \xrightarrow{\sigma} M' \quad N \xrightarrow{\sigma} N'}{M \mid N \xrightarrow{\sigma} M' \mid N'}$$

(Zero) is similar but regards empty networks. Rules (RcvPar) and (Sync) have their symmetric counterpart.

Let us explain the rules in Table 3 with an example.

Example 1. Consider the network

$$Net \overset{\text{def}}{=} k[!\langle v\rangle.?(x).P]^{\nu_k}_{0} \;\big|\; l[?(x).Q]^{\nu_l}_{0} \;\big|\; m[!\langle w\rangle]^{\nu_m}_{0} \;\big|\; n[?(y).R]^{\nu_n}_{0}$$

with the following communication topology: $\nu_k = \{l, m, \hat{l}\}$, $\nu_l = \{k, m\}$, $\nu_m = \{k, l, n, \hat{l}, \hat{m}\}$, and $\nu_n = \{m\}$. There are two possible broadcast communications originating from stations k and m, respectively. Let us suppose k starts broadcasting. By applying rules (Snd), (Rcv), (Exp), (OutRng), (RcvPar), and (Sync) we have:

$$\begin{array}{rl} Net \xrightarrow{k!v} & k[\langle v\rangle^{\delta_v}.?(x).P]^{\nu_k}_{0} \;\big|\; l[(x)_{k:v}.Q]^{\nu_l}_{\delta_v} \;\big|\; m[!\langle w\rangle]^{\nu_m}_{\delta_v} \;\big|\; n[?(y).R]^{\nu_n}_{0} \\ = & Net_1 \ . \end{array}$$

Now, by maximal progress, the station m must start transmitting at the same instant of time. Supposing $\delta_v < \delta_w$ we have:

$$\begin{array}{rl} Net_1 \xrightarrow{m!w} & k[\langle v\rangle^{\delta_v}.?(x).P]^{\nu_k}_{\delta_w} \;\big|\; l[(x)_{\bot}.Q]^{\nu_l}_{\delta_w} \;\big|\; m[\langle w\rangle^{\delta_w}]^{\nu_m}_{\delta_v} \;\big|\; n[(y)_{m:w}.R]^{\nu_n}_{\delta_w} \\ = & Net_2 \ . \end{array}$$

Now, node l is exposed to a collision and its reception is doomed to fail. Notice that, although node m was already exposed when it started transmitting, node n will receive correctly the message w from m.

Let us comment on rules of Table 4. Rules (Time-0) and (Time-t) model the passage of one time unit for non-exposed and exposed nodes, respectively. In both rules the exposure indicator is decreased. Notice that for $W = !\langle v\rangle.P$ none of these two rules can be applied, as for maximal progress no transmission can be delayed. Notice also that for $W = (x)_v.P$ and $t = 1$, by an application of rule (Time-t), the node evolves into $n[\{^{v}/_{x}\}P]^{\nu}_{0}$ which syntactically equals to

Table 5. LTS - Matching and recursion

$$\text{(Then)}\ \frac{n[P]^{\nu}_{t} \xrightarrow{\lambda} n[P']^{\nu}_{t'}}{n[[v=v]P,Q]^{\nu}_{t} \xrightarrow{\lambda} n[P']^{\nu}_{t'}} \qquad \text{(Else)}\ \frac{n[Q]^{\nu}_{t} \xrightarrow{\lambda} n[Q']^{\nu}_{t'} \quad v_1 \neq v_2}{n[[v_1=v_2]P,Q]^{\nu}_{t} \xrightarrow{\lambda} n[Q']^{\nu}_{t'}}$$

$$\text{(Rec)}\ \frac{n[\{^{\tilde{v}}\!/\!_{\tilde{x}}\}P]^{\nu}_{t} \xrightarrow{\lambda} n[P']^{\nu}_{t'} \quad H(\tilde{x}) \overset{\text{def}}{=} P}{n[H\langle\tilde{v}\rangle]^{\nu}_{t} \xrightarrow{\lambda} n[P']^{\nu}_{t'}}$$

$n[\{^{v}\!/\!_{x}\}P]^{\nu}_{0}$. Finally, notice that for $W = ?(x).P$ and $t > 0$ rule (Time-t) cannot be applied. In this case, we must apply rule (RcvFail) to model a failure in reception. This may happen, for instance, when the receiver misses the preamble starting a transmission, or when a receiver wakes up in the middle of an ongoing transmission. Again, we recall that we assumed the syntactic equality $m[(x)_v.P]^{\nu}_{0} = m[\{^{v}\!/\!_{x}\}P]^{\nu}_{0}$. Rule (Zero-$\sigma$) is straightforward. Rule (Par-σ) models time synchronisation. This is possible because our networks are connected. Rule (Par-σ) has its symmetric counterpart.

Example 2. Let us continue with the previous example. Let us show how the system evolves after δ_v and δ_w time units. We recall that $0 < \delta_v < \delta_w$. For simplicity let us define $\delta := \delta_w - \delta_v$:

$$\begin{array}{lllllllll}
Net_2 & \xrightarrow{\sigma}{}^{\delta_v} & k[?(x).P]^{\nu}_{\delta} & \mid & l[(x)_{\perp}.Q]^{\nu}_{\delta} & \mid & m[\langle w\rangle^{\delta}]^{\nu}_{0} & \mid & n[(y)_{m:w}.R]^{\nu}_{\delta} \\
& \xrightarrow{\sigma} & k[(x)_{\perp}.P]^{\nu}_{\delta-1} & \mid & l[(x)_{\perp}.Q]^{\nu}_{\delta-1} & \mid & m[\langle w\rangle^{\delta-1}]^{\nu}_{0} & \mid & n[(y)_{m:w}.R]^{\nu}_{\delta-1} \\
& \xrightarrow{\sigma}{}^{\delta-1} & k[\{^{\perp}\!/\!_{x}\}P]^{\nu}_{0} & \mid & l[\{^{\perp}\!/\!_{x}\}Q]^{\nu}_{0} & \mid & m[\mathsf{nil}]^{\nu}_{0} & \mid & n[\{^{m:w}\!/\!_{y}\}R]^{\nu}_{0}\ .
\end{array}$$

Notice that, after δ_v instants of time, node k will start a reception in the middle of an ongoing transmission (the transmitter being m). This will lead to a failure at k.

In the rest of the paper, the metavariable λ will range over the following labels: $m!v$, $m?v$, and σ. In Table 5 we report the obvious rules for nodes containing matching and recursive processes (we recall that only guarded recursion is allowed).

2.2 Well-Formedness

The syntax presented in Table 1 allows to derive inconsistent networks, i.e. networks that do not have a realistic counterpart. Below we give a number of definitions to rule out ill-formed networks.

As networks addresses are unique, we assume that there cannot be two nodes with the same name in the same network.

Definition 1 (Node uniqueness). *A network M is said to be* node-unique *if whenever $M \equiv M_1 \mid m[W_1]^{\nu}_{t} \mid n[W_2]^{\nu'}_{t'}$ it holds that $m \neq n$.*

We also assume network connectivity, i.e. all nodes are connected to each other, although not always directly. We recall that all nodes have the same transmission range. Formally,

Definition 2 (Network connectivity). *A network M is said to be* connected *if*

- *whenever $M \equiv N \mid m[W_1]^{\nu}_{t} \mid n[W_2]^{\nu'}_{t'}$ with $m \in \nu'$ it holds that $n \in \nu$;*
- *for all $m, n \in \mathsf{nds}(M)$ there is a sequence of nodes $m_1, \ldots, m_k \in \mathsf{nds}(M)$, with neighbouring $\nu_1, \ldots, \nu_k$, respectively, such that $m{=}m_1$, $n{=}m_k$, and $m_i \in \nu_{i+1}$, for $1 \leq i \leq k{-}1$.*

The next definition is about the consistency of exposure indicators of nodes. Intuitively, the exposure indicators of active senders and active receivers must be consistent with their current activity (transmission/reception). Moreover, the neighbours of active senders must have their exposure indicators consistent with the duration of the transmission.

Definition 3 (Exposure consistency). *A network M is said to be* exposure-consistent *if the following conditions are satisfied.*

1. *If $M \equiv N \mid m[(x)_v.P]^{\nu}_{t}$, then $t > 0$.*
2. *If $M \equiv N \mid m[(x)_v.P]^{\nu}_{t}$, with $v \neq \bot$, then $0 < t \leq \delta_v$.*
3. *If $M \equiv N \mid m[\langle v\rangle^r.P]^{\nu}_{t}$, then $0 < r \leq \delta_v$.*
4. *If $M \equiv N \mid m[\langle v\rangle^r.P]^{\nu}_{t} \mid n[W]^{\nu'}_{t'}$, with $m \in \nu'$, then $0 < r \leq t'$.*
5. *Let $M \equiv N \mid n[W]^{\nu}_{t}$ with $t{>}0$. If $\mathsf{active}(k, N) \neq t$ for all k in $\nu{\cap}\mathsf{actsnd}(N)$, then there is k' in $\nu \backslash \mathsf{nds}(N)$ such that whenever $N \equiv N' \mid l[W']^{\nu'}_{t'}$, with $k' \in \nu'$, then $t' \geq t$.*

The next definition is about the consistency of transmitting stations. The first and the second part are about successful transmissions, while the third part is about collisions.

Definition 4 (Transmission consistency). *A network M is said to be* transmission-consistent *if the following conditions are satisfied.*

1. *If $M \equiv N \mid n[(x)_v.Q]^{\nu}_{t}$ and $v \neq \bot$, then $\mid \mathsf{actsnd}(N) \cap \nu \mid \leq 1$.*
2. *If $M \equiv N \mid m[\langle w\rangle^r.P]^{\nu}_{t} \mid n[(x)_v.Q]^{\nu'}_{t'}$, with $m \in \nu'$ and $v \neq \bot$, then (i) $v = m{:}w$, and (ii) $r = t'$.*
3. *If $M \equiv N \mid n[(x)_v.P]^{\nu}_{t}$, with $\mid \mathsf{actsnd}(N) \cap \nu \mid > 1$, then $v = \bot$.*

Definition 5 (Well-formedness). *A network M is said to be* well-formed *if it is node-unique, connected, exposure-consistent, and transmission-consistent.*

In the sequel, we will work only with well-formed networks.

3 Properties

We start proving three desirable time properties of TCWS: time determinism, patience, and maximal progress. We then show that our LTS preserves network well-formedness.

Proposition 1 formalises the determinism nature of time passing: a network can reach at most one new state by executing the action σ.

Proposition 1 (Time Determinism). *Let M be a well-formed network. If $M \xrightarrow{\sigma} M'$ and $M \xrightarrow{\sigma} M''$ then M' and M'' are syntactically the same.*

Proof By induction on the length of the proof of $M \xrightarrow{\sigma} M'$. □

In [22,23], the maximal progress property says that processes communicate as soon as a possibility of communication arises. However, unlike [22,23], in our calculus message transmission requires a positive amount of time. So, we generalise the property saying that transmissions cannot be delayed.

Proposition 2 (Maximal Progress). *Let M be a well-formed network. If there is N such that $M \xrightarrow{m!v} N$ then $M \xrightarrow{\sigma} M'$ for no network M'.*

Proof Because sender nodes cannot perform σ-actions. □

The last time property is patience. In [22,23] patience guarantees that a process will wait indefinitely until it can communicate. In our setting, this means that if no transmissions can start then it must be possible to execute a σ-action to let time pass.

Proposition 3 (Patience). *Let M be a well-formed network. If $M \xrightarrow{m!v} M'$ for no network M' then there is a network N such that $M \xrightarrow{\sigma} N$.*

Proof By contradiction and then by induction on the structure of M. □

Finally, we prove that network well-formedness is preserved at runtime. In particular, the preservation of exposure- and transmission-consistency are the more interesting and delicate results.

Theorem 1 (Subject reduction). *If M is a well-formed network, and $M \xrightarrow{\lambda} M'$ for some label λ and network M', then M' is well-formed as well.*

Proof By transition induction. □

4 Observational Semantics

In this section we propose a notion of timed behavioural equivalence for our wireless networks. Our starting point is Milner and Sangiorgi's barbed congruence [24], a standard contextually-defined program equivalence. Intuitively, two terms are barbed congruent if they have the same *observables*, in all possible contexts, under all possible *evolutions*. The definition of barbed congruence strongly relies on two crucial concepts: a reduction semantics to describe how a system evolves, and a notion of observable which says what the environment can observe in a system.

From the operational semantics given in Section 2.1 it should be clear that a wireless network evolves transmitting messages. Notice that a transmission in a network does not require any synchronisation with the environment. Thus, we can define the reduction relation $\twoheadrightarrow$ between networks using the following inference rule

$$\text{(Red)}\ \frac{M \xrightarrow{m!v} N}{M \twoheadrightarrow N}\ .$$

We write $\twoheadrightarrow^\star$ for the reflexive and transitive closure of $\twoheadrightarrow$.

Now, let us focus on the definition of an appropriate notion of observable. In our calculus, as in CCS [2] and in π-calculus [4], we have both transmission and reception of messages. However, in broadcast calculi only the transmission of messages may be observed [25,10]. In fact, an observer cannot see whether a given node actually receives a broadcast value. In particular, if the node $m[!\langle v\rangle.P]^{\nu}_{t}$ evolves into $m[\langle v\rangle^r.P]^{\nu}_{t}$ we do not know whether some of the neighbours have actually synchronised for receiving the message v. On the other hand, if a *non-exposed* node $n[?(x).P]^{\nu}_{0}$ evolves into $n[(x)_{m:v}.P]^{\nu}_{t}$, then we can be sure that some node in ν has started transmitting. Notice that node n can certify the reception of a message v from a transmitter m only if it receives the whole message without collisions.

Following Milner and Sangiorgi [24] we use the term "barb" as synonymous of observable.

Definition 6 (Barbs). *Let M be a well-formed network. We write $M \downarrow_n$, if $M \equiv N \mid m[\langle v\rangle^r.P]^{\nu}_{t}$, for some m, v, r, P, t and ν, such that $n \in \nu$ and $n \notin$* $\mathsf{nds}(N)$*. We write $M \Downarrow_n$ if there is M' such that $M \twoheadrightarrow^\star M' \downarrow_n$.*

The barb $M \Downarrow_n$ says that there is a transmission at M reaching the node n of the environment. The observer can easily detect such a transmission placing a receiver with timeout at n. Say, something like $n[\lfloor ?(x).\mathbf{0}\rfloor !\langle w\rangle.\mathbf{0}]^{\nu}_{t}$, where $M \mid n[\lfloor ?(x).\mathbf{0}\rfloor !\langle w\rangle.\mathbf{0}]^{\nu}_{t}$ is well-formed, and $f \in \nu$, for some fresh name f. In this manner, if n is currently exposed to a transmission then, after a σ-action, the fresh barb at f is definitely lost. One may wonder whether the barb should mention the name m of the transmitter, which is usually recorded in some specific field of the packets. Notice that, in general, due to communication collisions, the observer may receive incomprehensible packets without being able to identify the transmitter. In fact, if $M \downarrow_n$ there might be different nodes of M which are currently transmitting to n. So, in our opinion, in our setting, it does not make sense to put the name of the transmitter in the barb.

Now, everything is in place to define our timed notion of barbed congruence. In the sequel, we write $\mathcal{R}$ to denote binary relations over well-formed networks.

Definition 7 (Barb preserving). *A relation $\mathcal{R}$ is said to be* barb preserving *if whenever $M\ \mathcal{R}\ N$ it holds that $M \downarrow_n$ implies $N \Downarrow_n$.*

Definition 8 (Reduction closure). *A relation $\mathcal{R}$ is said to be* reduction-closed *if $M \mathrel{\mathcal{R}} N$ and $M \mathrel{-\!\!\triangleright} M'$ imply there is N' such that $N \mathrel{-\!\!\triangleright}^{\star} N'$ and $M' \mathrel{\mathcal{R}} N'$.*

As we are interested in weak behavioural equivalences, the definition of reduction closure is given in terms of weak reductions.

Definition 9 (σ-closure). *A relation $\mathcal{R}$ is said to be σ*-closed *if $M \mathrel{\mathcal{R}} N$ and $M \xrightarrow{\sigma} M'$ imply there is N' such that $N \mathrel{-\!\!\triangleright}^{*} \xrightarrow{\sigma} \mathrel{-\!\!\triangleright}^{*} N'$ and $M' \mathrel{\mathcal{R}} N'$.*

When comparing two networks M and N, time must pass in the same manner for M and N.

Definition 10 (Contextuality). *A relation $\mathcal{R}$ is said* contextual *if $M \mathrel{\mathcal{R}} N$, for M and N well-formed, implies $M \mid O \mathrel{\mathcal{R}} N \mid O$ for all networks O such that $M \mid O$ and $N \mid O$ are well-formed.*

Finally, everything is in place to define timed reduction barbed congruence.

Definition 11 (Timed reduction barbed congruence). Timed reduction barbed congruence*, written $\cong$, is the largest symmetric relation over well-formed networks which is barb preserving, reduction-closed, σ-closed, and contextual.*

5 Bisimulation Proof Methods

The definition of timed reduction barbed congruence is simple and intuitive. However, due to the universal quantification on parallel contexts, it may be quite difficult to prove that two terms are equivalent. Simpler proof techniques are based on labelled bisimilarities. In this section, we propose an appropriate notion of bisimulation between networks. As a main result, we prove that our labelled bisimilarity is a proof-technique for timed reduction barbed congruence.

First of all we have to distinguish between transmissions which may be observed and transmissions which may not be observed.

$$(\mathrm{Shh})\ \frac{M \xrightarrow{m!v} N \quad \mathsf{ngh}(m,M)\subseteq\mathsf{nds}(M)}{M \xrightarrow{\tau} N} \qquad (\mathrm{Out})\ \frac{M \xrightarrow{m!v} N \quad \nu{:=}\mathsf{ngh}(m,M)\setminus\mathsf{nds}(M)\neq\emptyset}{M \xrightarrow{m!v\triangleright\nu} N}$$

Rule (Shh) models transmissions that cannot be detected by the environment. This happens if none of the potential receivers is in the environment. Rule (Out) models a transmission of a message that may be potentially received by the nodes ν of the environment. Notice that this transmission can be really observed at some node $n \in \nu$ only if no collisions arise at n during the transmission of v.

In the sequel, we use the metavariable α to range over the following actions: τ, σ, $m?v$, and $m!v\triangleright\nu$. Since we are interested in *weak behavioural equivalences*, that abstract over τ-actions, we introduce a standard notion of weak action: $\Rightarrow$ denotes the reflexive and transitive closure of $\xrightarrow{\tau}$; $\xRightarrow{\alpha}$ denotes $\Rightarrow\xrightarrow{\alpha}\Rightarrow$; $\xRightarrow{\hat{\alpha}}$ denotes $\Rightarrow$ if $\alpha = \tau$ and $\xRightarrow{\alpha}$ otherwise.

Definition 12 (Bisimilarity). *A relation $\mathcal{R}$ over well-formed networks is a* simulation *if $M\ \mathcal{R}\ N$ implies that*

- $\mathsf{nds}(M) = \mathsf{nds}(N)$
- *whenever $M \xrightarrow{\alpha} M'$ there is N' such that $N \overset{\hat{\alpha}}{\Longrightarrow} N'$ and $M'\ \mathcal{R}\ N'$.*

A relation $\mathcal{R}$ is called bisimulation *if both $\mathcal{R}$ and its converse are simulations. We say that M and N are* bisimilar*, written $M \approx N$, if there is some bisimulation $\mathcal{R}$ such that $M\ \mathcal{R}\ N$.*

The requirement that two bisimilar networks must have the same nodes is quite reasonable. Technically, this is necessary to prove that the bisimilarity is a congruence.

In order to prove that our labelled bisimilarity implies timed reduction barbed congruence we have to show its contextuality.

Theorem 2 ($\approx$ is contextual). *Let M and N be two well-formed networks such that $M \approx N$. Then $M \mid O \approx N \mid O$ for all networks O such that $M \mid O$ and $N \mid O$ are well-formed.*

Proof See the Appendix. □

Theorem 3 (Soundness). *Let* ngh *be a neighbouring function and M and N two well-formed networks wrt* ngh *such that $M \approx N$. Then $M \cong N$.*

Proof Contextuality follows from Theorem 2. Reduction and σ-closure follow by definition. As to barb preservation, it follows by Theorem 2 choosing $O \overset{\text{def}}{=} n[\lfloor ?(x).\mathbf{0} \rfloor !\langle w \rangle.\mathbf{0}]_t^\nu$ such that $M \mid O$ and $N \mid O$ are well-formed, and $f \in \nu$, for some fresh name f. □

Below, we report a number of algebraic properties on well-formed networks that can be proved using our bisimulation proof-technique.

Theorem 4. *1. $n[\mathsf{nil}]_t^\nu \approx n[Sleep]_t^\nu$, where $Sleep \overset{\text{def}}{=} \sigma.Sleep$.*

2. $n[\mathsf{nil}]_t^\nu \approx n[P]_t^\nu$, if P does not contain sender processes.

3. $n[\sigma^r.P]_s^\nu \approx n[\sigma^r.P]_t^\nu$ if $s \leq r$ and $t \leq r$.

4. $m[\langle v \rangle^r.P]_t^\nu \mid n[(x)_v.Q]_r^{\nu'} \approx m[\langle v \rangle^r.P]_t^\nu \mid n[\sigma^r.\{^v/_x\}Q]_r^{\nu'}$, if $\nu' = \{m\}$.

5. $m[!\langle v \rangle.P]_s^\nu \mid n[(x)_w.Q]_t^{\nu'} \approx m[!\langle v \rangle.P]_s^\nu \mid n[(x)_\perp.Q]_t^{\nu'}$, if $m \in \nu'$.

6. $m[!\langle v \rangle.P]_t^\nu \mid N \approx m[!\langle w \rangle.P]_t^\nu \mid N$, if for all $n \in \nu$ it holds that $N \equiv n[W]_{t'}^{\nu'} \mid N'$, with $t' \geq \mathsf{max}(\delta_v, \delta_w)$.

Proof By exhibiting the appropriate bisimulations. □

The first and the second law show different but equivalent nodes that do not interact with the rest of the network. The third law is about exposed and sleeping nodes. The forth law is about successful reception. The fifth law is about collisions: due to maximal progress the node m will start transmitting and the active receiver n is doomed to fail. The sixth law tells about the blindness of receivers exposed to collisions. In particular, if all neighbours of a transmitter are exposed, then the content of the transmission is irrelevant as all recipients will fail. Only the duration of the transmission may be important as the exposure indicators of the neighbours may change.

6 Future and Related Work

We have proposed a timed process calculus for IBBS networks with a focus on communication collisions. To our knowledge this is the first timed process calculus for wireless networks. In our model, time and collisions are treated in a completely *orthogonal* way.

We believe that our calculus represents a solid basis to develop *probabilistic theories* to do quantitative evaluations on collisions, and more generally on node failures. This will be one of the next directions of our research. We are also interested in using our timed calculus as a basis to develop *trust models* for wireless systems. Trust establishment in ad hoc networks is an open and challenging field. In fact, without a centralised trusty entity it is not obvious how to build and maintain trust. Nevertheless, the notion of time seems to be important to represent credentials' expiration.

For simplicity, in TCWS we assume a static network topology. As a consequence, our result mainly applies to *stationary networks*. Notice, movement is not relevant in important classes of wireless systems, most notably *sensor networks* (not all sensor networks are stationary, but the stationary case is predominant). We believe it is possible to extend our calculus to model disciplined forms of mobility along the lines of [13,14] in which neighbourings may change maintaining the network connectivity. This will be one of the next directions of our research.

Let us examine now the most relevant related work. Nanz and Hankin [8] have introduced a calculus for Mobile Wireless Networks (CBS$^\sharp$), relying on graph representation of node localities. The main goal of the paper is to present a framework for specification and security analysis of communication protocols for mobile wireless networks. Merro [10] has proposed a process calculus for Mobile Ad Hoc Networks with a labelled characterisation of reduction barbed congruence. Godskesen [11] has proposed a calculus for mobile ad hoc networks (CMAN). The paper proves a characterisation of reduction barbed congruence in terms of a contextual bisimulation. It also contains a formalisation of an attack on the cryptographic routing protocol ARAN. Singh, Ramakrishnan, and Smolka [9] have proposed the ω-calculus, a conservative extension of the π-calculus. A bisimulation in "open" style is provided. The ω-calculus is then used for modelling the AODV routing protocol. Ghassemi et al. [13] have proposed a process algebra for mobile ad hoc networks (RBPT) where, topology changes are implicitly modelled in the (operational) semantics rather than in the syntax. The authors propose a notion of bisimulation for networks parameterised on a set of topology invariants that must be respected by equivalent networks. This work in then refined in [14] where the authors propose an equational theory for an extension of RBPT. All the previous calculi abstract from the presence of interferences. Mezzetti and Sangiorgi [7] have instead proposed the CWS calculus, a lower level calculus to describe interferences in wireless systems. In their LTS there is a separation between transmission beginning and transmission ending. Our work was definitely inspired by [7].

None of the calculi mentioned above deals with time, although there is an extensive literature on timed process algebra. Aceto and Hennessy [26] have presented a simple process algebra where time emerges in the definition of a *timed observational equivalence*, assuming that beginning and termination of actions are distinct events which can be observed. Hennessy and Regan [22] have proposed a timed version of CCS. Our action σ takes inspiration from theirs. The authors have developed a semantic theory based on testing and characterised in terms of a particular kind of ready traces. Prasad [23] has proposed a timed variant of his CBS [27], called TCBS. In TCBS a time out can force a process wishing to speak to remain idle for a specific interval of time; this corresponds to have a priority. Corradini et al. [28] deal with *durational actions* proposing a framework relying on the notions of reduction and observability to naturally incorporate timing information in terms of process interaction. Our definition of timed reduction barbed congruence takes inspiration from theirs. Laneve and Zavattaro [29] have proposed a timed extension of π-calculus where time proceeds asynchronously at the network level, while it is constrained by the local urgency at the process level. They propose a timed bisimilarity whose discriminating is weaker when local urgency is dropped.

Acknowledgements. We thank Sebastian Nanz for a preliminary discussion on timed calculi for wireless networks, and Davide Quaglia for insightful discussions on the IEEE 802.11 standard. Many thanks to Matthew Hennessy and Andrea Cerone for their precious comments on a early draft of the paper.

References

1. IEEE 802.11 WG: ANSI/IEEE standard 802.11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. IEEE Computer Society, Los Alamitos (2007)
2. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
3. Bergstra, J., Klop, J.: Process algebra for synchronous communication. Information and Computation 60, 109–137 (1984)
4. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes (Parts I and II). Information and Computation 100, 1–77 (1992)
5. Cardelli, L., Gordon, A.: Mobile ambients. Theoretical Computer Science 240(1), 177–213 (2000)
6. Hennessy, M., Riely, J.: A typed language for distributed mobile processes. In: 25th POPL. ACM Press, New York (1998)
7. Mezzetti, N., Sangiorgi, D.: Towards a Calculus For Wireless Systems. Electronic Notes in Theoretical Computer Science 158, 331–353 (2006)
8. Nanz, S., Hankin, C.: A Framework for Security Analysis of Mobile Wireless Networks. Theoretical Computer Science 367(1-2), 203–227 (2006)
9. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A Process Calculus for Mobile Ad Hoc Networks. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 296–314. Springer, Heidelberg (2008)
10. Merro, M.: An Observational Theory for Mobile Ad Hoc Networks (full paper). Information and Computation 207(2), 194–208 (2009)

11. Godskesen, J.: A Calculus for Mobile Ad Hoc Networks. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 132–150. Springer, Heidelberg (2007)
12. Godskesen, J.: A Calculus for Mobile Ad-hoc Networks with Static Location Binding. Electronic Notes in Theoretical Computer Science 242(1), 161–183 (2009)
13. Ghassemi, F., Fokkink, W., Movaghar, A.: Restricted Broadcast Process Theory. In: SEFM, pp. 345–354. IEEE Computer Society, Los Alamitos (2008)
14. Ghassemi, F., Fokkink, W., Movaghar, A.: Equational Reasoning on Ad Hoc networks. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 113–128. Springer, Heidelberg (2010)
15. Mock, M., Frings, R., Nett, E., Trikaliotis, S.: Continuous Clock Synchronization in Wireless Real-Time Applications. In: SRDS, pp. 125–133. IEEE Computer Society, Los Alamitos (2000)
16. Ganeriwal, S., Kumar, R., Srivastava, M.: Timing-Sync Protocol for Sensor Networks. In: SenSys, pp. 138–149. ACM Press, New York (2003)
17. Sichitiu, M.L., Veerarittiphan, C.: Simple, Accurate Time Synchronization for Wireless Sensor Networks. In: WCNC, pp. 1266–1273. IEEE Computer Society, Los Alamitos (2003)
18. Su, W., Akyildiz, I.: Time-Diffusion Synchronization Protocols for Sensor Networks. IEEE/ACM Transactions on Networking 13(2), 384–397 (2005)
19. Li, Q., Rus, D.: Global Clock Synchronization in Sensor Networks. IEEE Transactions on Computers 55(2), 214–226 (2006)
20. Yoon, S., Veerarittiphan, C., Sichitiu, M.L.: Tiny-sync: Tight time synchronization for wireless sensor networks. ACM Transactions on Sensor Networks 3(2), 81–118 (2007)
21. Sundararaman, B., Buy, U., Kshemkalyani, A.D.: Clock synchronization for wireless sensor networks: a survey. Ad Hoc Networks 3(3), 281–323 (2005)
22. Hennessy, M., Regan, T.: A process algebra for timed systems. Information and Computation 117(2), 221–239 (1995)
23. Prasad, K.: Broadcasting in Time. In: Hankin, C., Ciancarini, P. (eds.) COORDINATION 1996. LNCS, vol. 1061, pp. 321–338. Springer, Heidelberg (1996)
24. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 685–695. Springer, Heidelberg (1992)
25. Rathke, J., Sassone, V., Sobocinski, P.: Semantic Barbs and Biorthogonality. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 302–316. Springer, Heidelberg (2007)
26. Aceto, L., Hennessy, M.: Towards action-refinement in process algebras. Information and Computation 103(2), 204–269 (1993)
27. Prasad, K.: A Calculus of Broadcasting Systems. Science of Computer Programming 25(2-3) (1995)
28. Corradini, F., Ferrari, G., Pistore, M.: On the semantics of durational actions. Theoretical Computer Science 269(1-2), 47–82 (2001)
29. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)

Model Checking Linear Duration Invariants of Networks of Automata*

Miaomiao Zhang[1], Zhiming Liu[2], and Naijun Zhan[3]

[1] School of Software Engineering, Tongji University, Shanghai, China
miaomiao@tongji.edu.cn
[2] International Institute of Software Technology,
United Nations University, Macau, China
Z.Liu@iist.unu.edu
[3] Lab. of Computer Science, Institute of Software, CAS, Beijing, China
znj@ios.ac.cn

Abstract. Linear duration invariants (LDIs) are important safety properties of real-time systems. In this paper, we reduce the problem of verification of a network of timed automata against an LDI to an equivalent problem of model checking whether a failure state is never reached. Our approach is first to transform each component automaton $\mathcal{A}_i$ of the network $\mathcal{A}$ to an automaton $\mathcal{G}_i$. The transformation helps us to record entry and exit to critical locations that appear in the LDI. We then introduce an auxiliary checker automaton $\mathcal{S}$ and define a failure state to verify the LDI on a given interval. Since a model checker checks exhaustively, a failure of the checker automaton to find the failure state will prove that the LDI holds.

1 Introduction

The invariants constructed from linear inequalities of integrated durations of system states are important properties of real-time systems. For example, in a container unloading system, the required property has the form "for any observation interval that is longer than 60 seconds, the idle time for a device is at most one twentieth of the time".

This kind of properties are often specified by *linear duration invariants* (LDIs) [13] of the following form:

$$A \leq \ell \leq B \Rightarrow \sum_{s \in S} c_s \textstyle\int s \leq M \tag{1}$$

where $\int s$ is the duration of state s, A, B, c_s and M are real numbers. The duration $\int s$ of state s and the length ℓ are mappings from time intervals to reals. For an observation time interval $[b, e]$, $\int s$ defines the accumulated time for the presence of state s over $[b, e]$ and ℓ is the length $e - b$ of the interval. An LDI $\mathcal{D}$ simply says that for any observation time interval $[b, e]$, if the length ℓ of the interval satisfies the constraint $A \leq \ell \leq B$ then

* The work is partly supported by the projects NSFC-60603037, NSFC-90718014, NSFC-60721061, NSFC-60573007, NSFC-90718041, NSFC-60736017, STCSM No.08510700300, and HighQSoftD and HTTS funded by Macao S&TD Fund.

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 244–259, 2010.

the durations of the system states over that interval should satisfy the linear constraint $\sum_{s\in S} c_s \int s \leq M$. We use $\Sigma(\mathcal{D})$ to denote the sum of the durations $\sum_{s\in S} c_s \int s$.

In this paper we consider the problem of automatic verification of a network of timed automata against an LDI, where each automaton is *closed* and *diagonal-free*. To address the issue, several algorithms have been proposed in the literature e.g. [3,10]. Different from the existing methods, in this paper, we develop a technique to reduce the problem of verification of a network of timed automata against an LDI to an equivalent problem of model checking whether a failure state cannot be reached. This will allow us to use existing model checkers, such as UPPAAL, to check the LDI. Our approach is first to transform each automaton $\mathcal{A}_i$ of the network $\mathcal{A}$ to an automaton $\mathcal{G}_i$. The transformation helps us to record entry and exit to critical locations that appear in the LDI.

Then we introduce an auxiliary timed automaton $\mathcal{S}$ from $\mathcal{A}$ and the LDI. $\mathcal{S}$ is used to calculate the observation time and the sum $\Sigma(\mathcal{D})$. In $\mathcal{S}$ we use a variable gc to record observation time, and another variable d to calculate the durations of system states. To approach the goal, the timed automaton $\mathcal{S}$ is constructed in different ways according to whether the constant B in (1) is finite or not. Subsequently, we define a failure state in $\mathcal{S}$ from the LDI $\mathcal{D}$, and prove that $\mathcal{D}$ is satisfied by $\mathcal{A}$ iff the failure state is never reached.

The rest of the paper is organized as follows. Section 2 recalls some basic notions of timed automaton and Duration Calculus. The main technical contribution is presented in Section 3. We present algorithms on how to construct the transformed automata $\mathcal{G}_i$ from an LDI and $\mathcal{A}_i$, and the two kinds of automata $\mathcal{S}$ respectively corresponding to the cases when B is finite and when B is infinite, and prove the main theorems. A case study is given in Section 4 to illustrate our technique. Section 5 gives a comparison between our approach and related work, discusses future work and concludes the paper.

2 Preliminaries

In this section, we introduce the notions that will be used later including the modelling language of UPPAAL, and Linear Duration Invariants (LDIs) defined in DC.

2.1 The Modelling Language

We first recall the notion of timed automata given in [1,2]. A timed automaton is a finite state machine equipped with a set of clocks. We use a set X of real value variables to represent the clocks and let $\Phi(X)$ be the set of clock constraints on X, which are conjunctions of the formulas of the form $x \leq c$ or $c \leq x$, where $x \in X$ and $c \in \mathbf{N}$. Formally,

Definition 1. *A timed automaton $\mathcal{A}$ is a tuple $\mathcal{A} = (L, l_0, \Sigma, X, E, I)$, where*

- *L is a finite set of locations,*
- *$l_0 \in L$ is the initial location,*
- *Σ is a finite set of actions, co-actions and the internal τ action,*
- *X is a finite set of clocks,*

- *I is a mapping that assigns each location $l \in L$ with a clock constraint $I(l) \in \Phi(X)$ called the invariant at l.*
- *$E \subseteq L \times \Phi(X) \times \Sigma \times 2^X \times L$ is a relation among locations, whose elements are called edges and annotated with an action, a guard and a set of clocks to be reset.*

Network of timed automata. A set of N timed automata $\mathcal{A}_i = (L_i, l_{i0}, \Sigma_i, X_i, E_i, I_i)$, $i = 1 \ldots N$, on the same sets of clocks and actions, are often composed into a network $\mathcal{A}$. A location for the network $\mathcal{A}$ is a vector $l = (l_1, \ldots, l_i, \ldots, l_N) \in L_1 \times \ldots \times L_N$ with the invariant $I(l) = \wedge_i I_i(l_i)$.

Binary synchronisation. Channels are declared as `chan` C. An edge labelled with $C!$ is synchronized with another labelled with $C?$. A synchronized pair is chosen nondeterministically if several combinations are enabled.

Priorities on processes. We follow the definition of priority of processes given in UPPAAL [15]. Process priorities are specified on the system level, using the separator $<$ to indicate that the process at the right hand has a higher priority to the one at the left side. If an instance of a template set is listed, all processes in the set will have the same priority.

The semantics of timed automata, networks of timed automata, channels are given in [15] and the reference file of the UPPAAL tool is also referred to [15].

2.2 Duration Calculus and Linear Duration Invariants

Duration calculus DC [12] is a logic for reasoning about durations of states of real-time systems. A comprehensive introduction to DC is given in the monograph by Zhou and Hansen [14]. In DC, a state s is *interpreted* as a function from the time domain $\mathbf{R}^+$ to the boolean values $1, 0$, and s is 1 at time t if the system is in state s at the time point and 0 otherwise. the duration of state s over the time interval $[b, e]$ is defined as the integral $\int_b^e s(t)dt$, which is exactly the accumulated present time of s in the interval $[b, e]$. ℓ is used to denote the length of considered interval, which is defined by $\int 1$. We have the following measure laws on durations[1]:

1. $0 \leq \int s \leq \ell$
2. $\int \neg s = \ell - \int s$
3. $\int s_1 \vee s_2 = \int s_1 + \int s_2 - \int s_1 \wedge s_2$

We consider the set of DC models that corresponds to all the behaviors of the network of timed automata $\mathcal{A}$. A behavior ρ of $\mathcal{A}$ is of the form $(S_0, t_0)(S_1, t_1) \ldots$, where each S_i is called a state of $\mathcal{A}$ which is a subset of the state variables of $\mathcal{A}$ and t_is are incremental, i.e. $t_i \leq t_{i+1}$ for any $i \in \mathbf{N}$. Each behaviour defines an interpretation $\mathcal{I}$ of the DC formulas over the state variables of $\mathcal{A}$: for any state variable s of $\mathcal{A}$, $\mathcal{I}_s(t) = 1$ iff $\exists i \cdot (s \in S_i \wedge t \in [t_i, t_{i+1}])$. We also denote such $\mathcal{I}$ by $(\overline{s}, \overline{t})$ where $\overline{s} = (S_0, S_1, \ldots)$ and $\overline{t} = (t_0, t_1, \ldots)$ are respectively the sequence of states S_i and the sequence of time

[1] There are six axioms on durations (see [14]), but here we just list some of them which are used in this paper.

stamps t_i of ρ. Hence, $(\overline{s}, \overline{t}, [b, e])$ is a DC model representing an observation of $\mathcal{A}$ in the time interval $[b, e]$. We also call $(\overline{s}, \overline{t}, [b, e])$ an $\mathcal{A}$-model of DC.

For a given network of timed automata $\mathcal{A}$, we define the set of $\mathcal{A}$-models of DC with integral observation intervals [11] as

$$\mathcal{M}_I(\mathcal{A}) \mathrel{\widehat{=}} \{\sigma \mid \sigma = (\overline{s}, \overline{t}, [b, e]) \in \mathcal{M}(\mathcal{A}) \text{ and } b, e \in \mathbf{N}, b \leq e\}$$

Linear duration invariants. A *linear duration invariant* (LDI) of a network of timed automata $\mathcal{A}$ is a DC formula $\mathcal{D}$ of the form

$$A \leq \ell \leq B \Rightarrow \sum_{s \in L} c_s \textstyle\int s \leq M$$

where A, B, c_s and M are real numbers.

An LDI $\mathcal{D}$ is evaluated in an $\mathcal{A}$-model $(\mathcal{I}, [b, e])$ to tt, denoted by $(\mathcal{I}, [b, e]) \models \mathcal{D}$, iff $A \leq e - b \leq B \Rightarrow \sum_{s \in L} c_s \int_b^e \mathcal{I}_s(t)dt \leq M$ holds. $\mathcal{D}$ is satisfied by $\mathcal{A}$, denoted by $\mathcal{A} \models \mathcal{D}$, if $(\mathcal{I}, [b, e]) \models \mathcal{D}$ holds for all $\mathcal{A}$-models $(\mathcal{I}, [b, e])$. We use $\Sigma(\mathcal{D})$ to denote the the sum of the durations $\sum_{s \in L} c_s \int s$.

Digitization of LDIs w.r.t. timed automaton. Henzinger et al. [5] studied the question of which real-time properties can be verified by considering system behaviours featuring only integer durations. These results are applied to timed automata in [7], and it is shown that an approach using digital clocks is applicable to the verification of closed, diagonal-free timed automata. The digitization of duration calculus has also been studied in [6,8,9]. As to the digitization of an LDI, the following theorem has been proved in [11], where $\mathcal{A}$ is closed and diagonal-free.

Theorem 1. $\mathcal{M}(\mathcal{A}) \models \mathcal{D} \Leftrightarrow \mathcal{M}_I(\mathcal{A}) \models \mathcal{D}$.

Therefore only the set of integral $\mathcal{A}$ models of DC are studied in [11]. In the rest of the paper, we also only consider model $\sigma = (\overline{s}, \overline{t}, [b, e]) \in \mathcal{M}_I(\mathcal{A})$ that represents an observation of an integral behavior of $\mathcal{A}$ (i.e behavior in which transitions take place only at integer time) from an integer time point to an integer time point. So we will restrict $\mathcal{A}$ to be an integer-time model.

3 Verification of LDIs

In this section, we present our technique to reduce the verification of the satisfaction of an LDI $\mathcal{D}$ by a network of timed automata $\mathcal{A}$ to checking the property whether a failure state cannot be reached. In what follows, each automaton $\mathcal{A}_i$ is referred to an integer-time model. We start with the calculation of the duration of a location of the composite automaton of the network, i.e. a location vector.

3.1 Duration of a Location Vector s_j

Let $\mathcal{A}$ be a network of N timed automata $\mathcal{A}_i = (L_i, l_{i0}, \Sigma_i, X_i, E_i, I_i)$, $i = 1, \ldots, N$. A location of $\mathcal{A}$ is a vector $l = (l_1, \ldots, l_i \ldots l_N)$, where $l_i \in L_i$.

A location can be seen as a state variable. In the following, we require that each state expression s_j of a duration term $\int s_j$ in the LDI $\mathcal{D}$ is constructed by using logical connectives from the locations l_i of the component automata $\mathcal{A}_i$. For example, $l_1 \wedge l_2$ asserts that $\mathcal{A}$ is in a location where $\mathcal{A}_1$ and $\mathcal{A}_2$ are respectively in location l_1 and l_2. We are particularly interested in those state expressions s_j of the form $l_{a_1} \wedge \ldots \wedge l_{a_k}$, where $l_{a_i} \in L_{a_i}$ for $\{a_1, \ldots, a_k\} \subseteq \{1, \ldots, N\}$. We represent such a state expression by a *vector with free locations* $f \in (L_1 \cup \{\times_1\}) \times \ldots \times (L_N \cup \{\times_N\})$, such that $f[a_i] = l_{a_i}$and $f[b] = \times_b$ for $b \in \{1, \ldots, N\} \setminus \{a_1, \ldots, a_k\}$. This defines the set of vectors whose bth component $\times_b$ can be any location of $\mathcal{A}_b$.

Using the axioms in Subsection 2.2, it is easy to equivalently transform a general LDI to an LDI in which all state expressions can be represented by either a full location vector $\mathcal{A}$ or a vector with free locations. Thus, in the rest of the paper we only consider LDIs of this form.

We call each s_j that appears in an LDI *critical* location vector of $\mathcal{A}$ and use K to denote the number of *critical* location vectors in the LDI. A location l_i of an automaton $\mathcal{A}_i$ is called *p-critical* location of s_j if l_i occurs in s_j as a non-free location and $\mathcal{A}_i$ is a *critical* automaton of s_j. A location l_i of an automaton $\mathcal{A}_i$ is called *p-critical* location if it is a *p-critical* location of some s_j. We use W to denote the number of *p-critical* locations.

Example 1. Consider the following LDI

$$c_{s_1}\int(l_{11}, l_{22}, l_{31}) + c_{s_2}\int(\times_1, \times_2, l_{31}) + c_{s_3}\int(\times_1, l_{21}, l_{32}) \ \leq M$$

Its *critical location vectors* are respectively $s_1 = (l_{11}, l_{22}, l_{31})$, $s_2 = (\times_1, \times_2, l_{31})$ and $s_3 = (\times_1, l_{21}, l_{32})$, while its *p-critical locations* are l_{11}, l_{22}, l_{31}, l_{21} and l_{32}. Obviously, in this example, $K = 3$ and $W = 5$.

For a *critical* location vector s_j of $\mathcal{A}$, with the elapse of one time unit, $\mathcal{A}$ stays in s_j for one time unit if each automaton $\mathcal{A}_i$ *critical* to s_j stays in the *p-critical* location of s_j for one time unit. This one unit delay of $\mathcal{A}$ in s_j causes an increase of c_{s_j} to the sum $\Sigma(\mathcal{D})$ of the LDI.

Main technique. With the above definitions, the main idea of the technique can be sketched as follows:

- Firstly, we construct a network of automata $\mathcal{G}$ from $\mathcal{A}$ and $\mathcal{D}$ to record the entry and exit of the *p-critical* locations.
- As we need to check from any reachable state, whenever the antecedent of the LDI is true implies that its conclusion holds, we introduce $\mathcal{S}$ to count the observation time gc and the sums of the durations of the *critical* location vectors d from any reachable state.
- Finally, we construct a failure state such that $\mathcal{A} \models \mathcal{D}$ if and only if this failure state is never reached in $\mathcal{S}$.

3.2 Transformation of the Network of Automata

The network of automata $\mathcal{A}$ is first transformed to another network of automata to record the entry and exit of the *p-critical* locations. For this, we need to introduce a Boolean array $active$ with size W to indicate whether the *p-critical* locations are entered. The index k of this array denotes the $(k+1)^{th}$ *p-critical* location in the LDI. Initially the value of $active[k]$ is 0. It is set to 1 when the $(k+1)^{th}$ *p-critical* location is entered, and set to 0 when this location is exited.

We transform each component automaton $\mathcal{A}_i$ to an automaton $\mathcal{G}_i$. $\mathcal{G}_i$ is similar to $\mathcal{A}_i$ except that for the entry and exit of each *p-critical* location, the value of the corresponding $active$ is updated by 1 and 0 respectively. Note here if the initial location of $\mathcal{A}_i$ is the $(k+1)^{th}$ *p-critical* location, then an additional urgent location is introduced in $\mathcal{G}_i$ to set the value of $active[k]$ to 1. We call $\mathcal{G} = (\mathcal{G}_1, \| \dots, \| \mathcal{G}_N)$ constructed by this procedure the *network of transformed automata* of $\mathcal{A}$ for $\mathcal{D}$.

Example 2. Fig.1 gives a case of the transformation from $\mathcal{A}_1$ to $\mathcal{G}_1$, where `l0` is the first *p-critical* location and `l2` is the third *p-critical* location. An additional urgent location is introduced to set the value of $active[0]$ to 1.

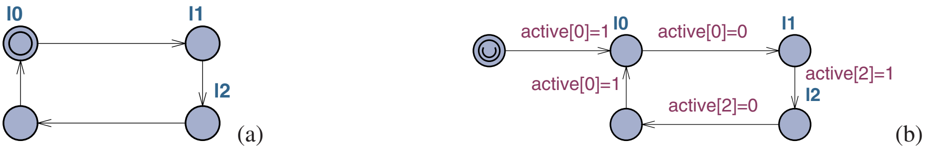


Fig. 1. Transformation from $\mathcal{A}_1$ to $\mathcal{G}_1$ a) $\mathcal{A}_1$ b) $\mathcal{G}_1$

3.3 Construction of the Auxiliary Automaton $\mathcal{S}$

In order to check whether the LDI is satisfied, we check for any path from a reachable state that the sum of the durations of the *critical location vectors* does not exceed M within the time interval $[A, B]$. For this, we build an auxiliary automaton $\mathcal{S}$, where the following variables are introduced and initialized to 0.

- gc is a local variable in $\mathcal{S}$ to record the length of an observation interval from s_r in a path of $\mathcal{G}$, and
- x is a local clock variable in $\mathcal{S}$ to record the elapse of one time unit, and
- d is a local variable in $\mathcal{S}$ to record the sum of the durations of the *critical* location vectors, i.e. the value $\Sigma(\mathcal{D})$.

In the construction of $\mathcal{S}$, to bound the value of d and gc, we need to use different methods depending on the constant B in an LDI is finite or infinite. For the update of variable gc, we introduce $B+1$-normalization when B is finite and A-normalization when B is infinite. Here A is the other constant in the antecedent of the LDI.

Definition 2. *($B+1$-normalization)*

$$norm_{B+1}(gc) = \begin{cases} gc+1, & \text{if } gc \leq B \\ B+1, & \text{if } gc > B \end{cases}$$

The intuition is that gc records the length of the current observation interval, and therefore the LDI $\mathcal{D}$ is satisfied trivially whenever $gc > B$. Hence, we do not need to record all the values of gc that are bigger than B. It is sufficient to record $B+1$ when the length of the observation time exceeds B.

Definition 3. *(A-normalization)*

$$norm_A(gc) = \begin{cases} gc + 1, & \text{if } gc < A \\ A, & \text{if } gc \geq A \end{cases}$$

Intuitively, the A normalization is dual to the B-normalization. With this normalization, for checking LDI $\mathcal{D}$ when gc equals A, we only need to check whether there exists a path along which the value of $\Sigma(\mathcal{D})$ is bigger than M.

In both cases (when B is finite and B is infinite), we require that the process $\mathcal{S}$ has higher priority than any other process. This is declared by `system` $\mathcal{G}_1, \ldots, \mathcal{G}_N < \mathcal{S}$, which means that at a given time-point, a transition in $\mathcal{G}_i$ is enabled only if all transitions of $\mathcal{S}$ are disabled.

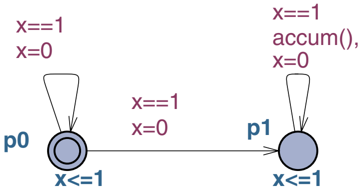


Fig. 2. Auxiliary automaton $\mathcal{S}$

When B is finite. Fig. 2 shows the automaton $\mathcal{S}$ when B is finite. There are two locations `p0` and `p1` and initially $\mathcal{S}$ stays in `p0`. The trick that $\mathcal{S}$ will nondeterministically stay in `p0` for any number of time units before moving to `p1` ensures that gc and d will start to count from any reachable state of $\mathcal{A}$.

In location `p1` of $\mathcal{S}$, with the elapse of one time unit, the values of gc and d are updated. These are implemented by the function `accum`() given in Fig 3, where we still use s_j to denote a *critical* location vector and K to denote the number of *critical* location vectors. In `accum`(), the first assignment assigns gc the value of $gc + 1$ if $gc \leq B$ and the value $B + 1$ otherwise, i.e. it is the implementation of the $B + 1$ normalization.

Note that with one time unit elapsed, $\mathcal{A}$ may not stay in any of the *critical* location vectors or may stay in several *critical* location vectors during the time unit. Therefore, function `accum`() uses a "for" loop to handle the latter case so that the update of d is correct. By the definitions of Subsection 3.1, checking if the duration of a *critical* location vector s_j is 1 over the one time unit interval is equivalent to checking if the duration of each *p-critical* location of this vector is 1. The following theorem is used to decide if the duration of a *p-critical* location is 1.

Theorem 2. *Let $\mathcal{G}$ be the network of transformed automata of $\mathcal{A}$ and $\mathcal{S}$ be the auxiliary automaton of $\mathcal{A}$ defined above. With one time unit elapsed when function accum() is executed, for the k^{th} p-critical location l, if $active[k-1] = 1$ then the duration of l is 1 over this one time unit interval, otherwise the duration of l is 0.*

Proof. Since each component automaton $\mathcal{A}_i$ we only consider its integer-time models, each discrete transition of the transformed automaton $\mathcal{G}_i$ is therefore taken at integer time point too. As observed from $\mathcal{S}$, in location `p1` function `accum`() is enabled each one time unit. By the declaration of process priority: `system` $\mathcal{G}_1, \ldots, \mathcal{G}_N < \mathcal{S}$, we have that any transition that is enabled to exit or enter a *p-critical* location must be executed after the execution of `accum`().

Let l be the k^{th} *p-critical* location of $\mathcal{G}_i$. Let τ_1 be the transition that enters the *p-critical* location l with the assignment $active[k-1] = 1$ and τ_2 be the transition that exits location l with the assignment $active[k-1] = 0$. In location `p1`, consider the one time unit interval $II = [\mathcal{S}.x = 0, \mathcal{S}.x = 1]$, where x is the local clock in $\mathcal{S}$. At time point $\mathcal{S}.x = 1$, function `accum`() is executed before any other enabled transition to check the duration of l over the interval II. To do this, it checks the value of $active[k-1]$.

- When $active[k-1] = 1$, suppose the duration of l is not 1 over the interval II, that is, it is either 0 or lies in the interval $(0, 1)$. In the former case, it implies that at the time point $\mathcal{S}.x = 1$, τ_1 is taken before `accum`(), which violates the assumption of process priority. In the latter case, it means that the time point that the action τ_1 takes place lies in the interval $II' = (\mathcal{S}.x = 0, \mathcal{S}.x = 1)$. This also contradicts the fact that $\mathcal{G}_i$ is an integer-time model. Therefore, when $active[k-1] = 1$, the duration of l is 1 over II.
- When $active[k-1] = 0$, suppose the duration of l is not 0 over the interval II, that is, it is either 1 or lies in the interval $(0, 1)$. If it is 1, then it must be the case that $\mathcal{G}_i$ stays in l for one time unit and τ_2 is executed before `accum`(). This also violates the assumption of process priority. If the duration of l is in the interval $(0, 1)$, then the time point τ_2 takes place lies in the interval $II' = (\mathcal{S}.x = 0, \mathcal{S}.x = 1)$. However this kind of transition is not allowed in $\mathcal{G}_i$. So when $active[k-1] = 1$, the duration of l is 0 over II.

We conclude the proof. □

The above theorem allows the calculation of the duration of a *critical* location vector in terms of the information of entry or exit of its *p-critical* locations. Obviously, if the duration of a *critical* location vector s_j is 1 over one time unit interval, the value of d is increased by the value of the coefficient of this vector, i.e, c_{s_j}. So the construction of $\mathcal{S}$ correctly records the durations of the *critical* location vectors from any reachable state of $\mathcal{A}$. In addition, all the variables introduced are bounded. This is because by assigning 0 to d when $gc > B$, the value of variable d is finite, also gc is bounded by $B + 1$.

```
void accum()
{     $gc = (gc \leq B?gc + 1 : B + 1)$
      for ($j = 1, j \leq K, j++$)
      { if each p-critical location of $s_j$ is entered
            $d = (gc \leq B?d + c_{s_j} : 0)$
      }
}
```

Fig. 3. Function `accum`() when B is finite

The corresponding failure state. The failure state $\mathcal{F}$ is $A \leq gc \leq B \wedge d > M$. We check $\mathcal{F}$ cannot be reached in $\mathcal{S}$. This property can be expressed in CTL [4] as

$$\psi_1 \mathrel{\widehat{=}} \texttt{A[] not } \mathcal{F} \tag{2}$$

We call $\mathcal{F}$ the *failure state* of $\mathcal{D}$ for $\mathcal{A}$.

Lemma 1. *Let $\mathcal{D}$ be an LDI of the network of timed automata $\mathcal{A}$, $\mathcal{G}$ the network of transformed automata of $\mathcal{A}$ for $\mathcal{D}$, $\mathcal{S}$ the auxiliary automaton of $\mathcal{A}$ for $\mathcal{D}$, $\mathcal{G} \parallel \mathcal{S}$ the parallel composition of G and S, $\mathcal{P}(\mathcal{G} \parallel \mathcal{S})$ the set of all paths of $\mathcal{G} \parallel \mathcal{S}$ and ψ_1 the failure state property. Then there exists a path $\rho_g \in \mathcal{P}(\mathcal{G} \parallel \mathcal{S})$ such that $\rho_g \not\models \psi_1$ iff there exists a path $\rho \in \mathcal{P}(\mathcal{A})$ such that $\rho \not\models \mathcal{D}$.*

Proof. From the construction procedure for $\mathcal{G}$ and $\mathcal{S}$, there is an obvious correspondence between a path ρ of $\mathcal{A}$ and a path ρ_g of $\mathcal{G} \parallel \mathcal{S}$ starting from the initial locations, that represents an observation of the system in the two models. Let $\ell(\rho)$ be the length of ρ, which represents the time of the observation, and $last(\rho_g)$ be the last node of ρ_g.

1. When $\ell(\rho) \leq B$, the value of gc at $last(\rho_g)$ equals $\ell(\rho)$, and the value of d at $last(\rho_g)$ is the value of the sum $\Sigma(\mathcal{D})$.
2. When $\ell(\rho) > B$, the value of gc at $last(\rho_g)$ is $B + 1$.

Consequently, the lemma follows immediately from the definition of the satisfaction relations $\models$ for LDIs and the definition of the failure state. □

Theorem 3. *Let $\mathcal{F}$ be the failure state of $\mathcal{D}$ for $\mathcal{A}$. When B is finite, $\mathcal{A} \models \mathcal{D}$ if and only if state $\mathcal{F}$ is never reached in $\mathcal{G} \parallel \mathcal{S}$.*

Construction example. Fig. 4 and Fig.5 give a case of the construction of the network of transformed automata $\mathcal{G}$ from $\mathcal{A}$ and the correspondent automaton $\mathcal{S}$. The LDI is of the form:

$$A \leq \ell \leq B \Rightarrow\ c_{s_1} \textstyle\int (l_j, l_m) + c_{s_2} \int (\times_1, l_n) \leq M$$

The *critical* location vectors are $s_1 = (l_j, l_m)$ and $s_2 = (\times_1, l_n)$. The first *p-critical* location is l_j, the second *p-critical* location is l_m and the third *p-critical* location is l_n.

When B is infinite. In terms of the automaton $\mathcal{S}$ constructed in the previous subsection, gc can increase infinitely and d could take an arbitrary value in case B is infinite. The above theorem is not applicable anymore. To bound the value of gc we will use "A-normalization" introduced before. Now we try to find a upper bound of d.

Definition 4. *A critical location vector s_p in $\mathcal{A}$ is said* positive *if $c_{s_p} > 0$. Let L_+ be the set of all positive critical location vectors in $\mathcal{A}$, l_i the ith p-critical location of a positive critical vector s_p and $\mathtt{u}(l_i)$ the maximum time units that $\mathcal{A}_i$ stays in l_i. We define $\mathtt{u}(s_p) = \mathtt{min}\{\mathtt{u}(l_1), \ldots, \mathtt{u}(l_i), \ldots, \mathtt{u}(l_N)\}$, and call $Q = \sum_{s_p \in L_+} (c_{s_p} \times \mathtt{u}(s_p))$ the maximum increment of $\mathcal{A}$.*

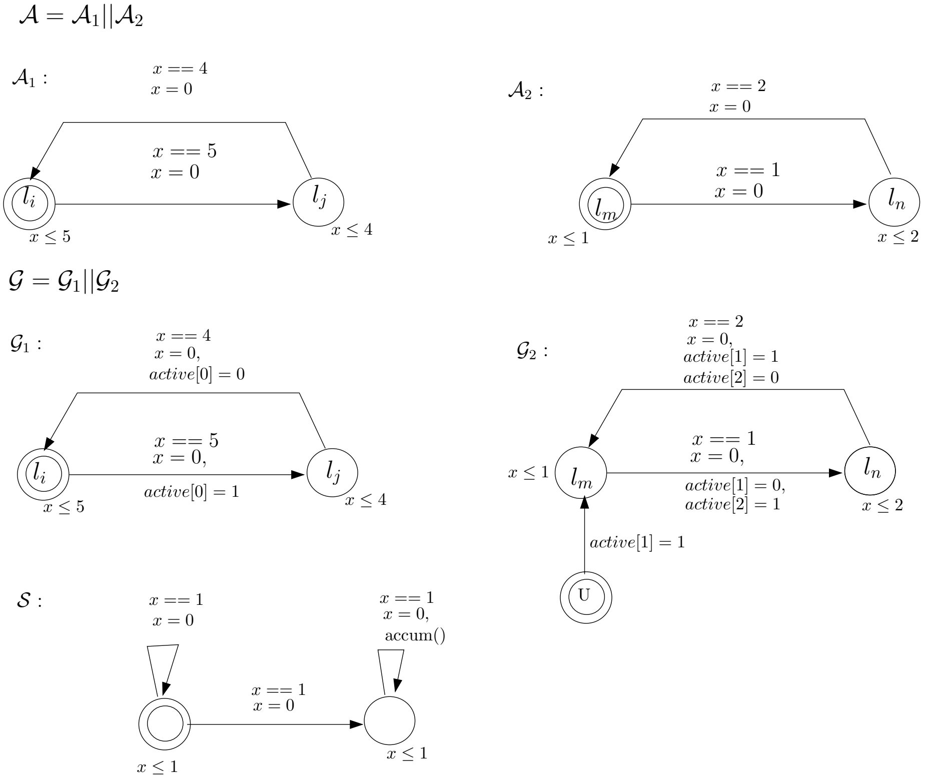


Fig. 4. The network of transformed automata $\mathcal{G}$ and the auxiliary automaton $\mathcal{S}$ in the example

```
void accum()
    {$gc = (gc \leq B?gc + 1 : B + 1)$
    if $active[0] \times active[1] == 1$
        $d = (gc \leq B?d + c_{s_1} : 0)$
    if $active[2] == 1$
        $d = (gc \leq B?d + c_{s_2} : 0)$}
```

Fig. 5. The update function `accum`() in the example

Note that $\mathsf{u}(s_p)$ is the *maximum* time that $\mathcal{A}$ stays in s_p because of the clock synchronization. This value is used in the calculation of Q, and Q is used to detect if a path of $\mathcal{A}$ contains a *positive loop* that takes non-zero time. If there is no *positive loop* in a path of $\mathcal{A}$, the value of d along that path can increase at most Q. In other words, if the value of d along a path increases more than Q, then there must be a positive loop in the path. It is in general difficult to calculate the actual value of $\mathsf{u}(s_p)$ as it requires all the $\mathsf{u}(l_i)$'s. So usually we calculate a value that is bigger than $\mathsf{u}(s_p)$ and assign it to $\mathsf{u}(s_p)$ when calculating Q.

Example. Suppose in the example shown in Fig. 4, $c_{s_1} > 0$ and $c_{s_2} > 0$, then we can let $Q = 4 \times c_{s_1} + 2 \times c_{s_2}$.

The auxiliary automaton $\mathcal{S}^+$ is similar to the one shown in Fig.2 except that the updates of gc and d are different. In other words, the function `accum()` is different. For any *critical* location vector s_j, we make the variable d bounded by updating it in different ways depending on whether the coefficient c_{s_j} of $\int s_j$ in $\Sigma(\mathcal{D})$ is negative or not.

1. The update of gc is done by the *A-normalization*.
2. The update of d is done according to the following rules:
   - if s_j has a non-negative coefficient c_{s_j}, then if $gc \geq A \wedge d > M$ then d is updated to $M + 1$; otherwise updated to $d + c_{s_j}$;
   - if s_j has a negative coefficient c_{s_j}, then if $gc \geq A \wedge d < M - Q$ then d keeps unchanged, otherwise d is updated to $d + c_{s_j}$.

When c_{s_j} is non-negative, if $gc \geq A$ and $d > M$, by setting d to $M + 1$, the value of d is finite. Moreover, when $gc \geq A$, gc remains as A, so gc is a bounded variable. Since the states that satisfy $gc \geq A \wedge d = M + 1$ imply $\mathcal{G} \parallel \mathcal{S}^+ \not\models \mathcal{D}$, it is obvious that the update does not change the verification result.

If c_{s_j} is negative, the update of d is done by setting $d = (gc \geq A \wedge d < M - Q ? d : d + c_{s_i})$. It is not hard to see why we set d to $d + c_{s_i}$ if $\neg(gc \geq A \wedge d < M - Q)$: we have to evaluate the value of d precisely when we do not have enough information for verifying if $\mathcal{D}$ is satisfied.

Now we prove that if $gc \geq A \wedge d < M - Q$, d keeping unchanged does not change the checking result of the LDI. To the end, we define another graph $\mathcal{S}^\bullet$ that is the same as $\mathcal{S}^+$ except that if $gc \geq A \wedge d < M - Q$ the assignment for d is $d = d + c_{s_j}$ in function `accum()`.

Similar to the case when B is finite, we define the*failure state* $\mathcal{F}'$: $gc \geq A \wedge d > M$. Also, we use CTL to express that $\mathcal{F}'$ is never reached.

$$\psi_2 \mathrel{\widehat{=}} \texttt{A[] not } \mathcal{F}' \tag{3}$$

Lemma 2. *Let $\mathcal{P}(\mathcal{G} \parallel \mathcal{S}^+)$ be the set of pathes of $\mathcal{G} \parallel \mathcal{S}^+$. There exists a path $\rho \in \mathcal{P}(\mathcal{G} \parallel \mathcal{S}^+)$ such that $\rho \not\models \psi_2$ if and only if there exists a path $\rho' \in \mathcal{P}(\mathcal{G} \parallel \mathcal{S}^\bullet)$ such that $\rho' \not\models \psi_2$.*

Proof. Notice that the topological structure of $\mathcal{S}^+$ and $\mathcal{S}^\bullet$ are the same. Each path $\rho = s_0^+, \ldots, s_m^+$ in $\mathcal{G} \parallel \mathcal{S}^+$ corresponds to exactly one path $\rho^\bullet = s_0^\bullet, \ldots, s_m^\bullet$ in $\mathcal{G}^\bullet \parallel \mathcal{S}$. Let s_i^+ and $s_i^\bullet$ be any two corresponding nodes respectively in ρ and $\rho^\bullet$. Then the value of gc at vertex s_i^+ is the same as the value of that at vertex $s_i^\bullet$. Due to the different updates of d in ρ and $\rho^\bullet$ for the negative coefficient of a vertex, we know that at vertex s_i^+, the value of d is bigger than or equal to the value of d at $s_i^\bullet$. Hence, if a path $\rho' = \rho^\bullet$ in $\mathcal{G} \parallel \mathcal{S}^\bullet$ does not satisfy ψ_2, then its corresponding path ρ in $\mathcal{G} \parallel \mathcal{S}^+$ does not satisfy ψ_2.

To prove the other direction, let ρ in $\mathcal{G} \parallel \mathcal{S}^+$ be such that $\rho \not\models \psi_2$ and ρ starts from the initial location. If $\rho^\bullet \not\models \psi_2$, we are done. Otherwise, we need to show that there will be a "positive cycle" in ρ, i.e. there is a cycle such that going along the cycle will increase the value of d properly by at least 1. We now give the illustration for the case $\rho \not\models \psi_2 \wedge \rho^\bullet \models \psi_2$. This case denotes that the values of d on ρ and on $\rho^\bullet$ are different and there should be a first node s_j^+ along ρ where the condition $gc \geq A \wedge d < M - Q \wedge c_{s_j} < 0$ holds. Thus, from s_j^+, the value of d is increased by at least $Q + 1$ to make $\rho \not\models \psi_2$.

From the definition of Q, in ρ there must be a "positive cycle" along which d will be increased by at least 1. From the correspondence relation between ρ and $\rho^\bullet$, $\rho^\bullet$ must also have a positive cycle $\mathcal{C}$. Thus ρ' is formed by increasing the number of repetition of the cycle $\mathcal{C}$ in $\rho^\bullet$, such that $\rho' \not\models \psi_2$. □

Therefore we conclude that d is a bounded integer variable. We now have another main theorem.

Theorem 4. *When B is infinite, $\mathcal{A} \models \mathcal{D}$ if and only state $\mathcal{F}'$ cannot be reached in $\mathcal{G} \parallel \mathcal{S}^+$.*

4 Case Study

We now use a simplified automated container system to illustrate our techniques. We assume there are infinite number of containers to be transported from a ship to a yard. One quay crane (QC) and two track cars (TC) are used to unload these containers. As to a container in the ship, the QC first transports it to an idle TC, then the TC delivers it to the yard. It takes 5 time units for a QC to move down and pick up a container, then it will wait until one of the TCs is idle. If either of the TCs is idle, the QC spends 3 time units unloading the container to the idle TC, and 10 time units to get back to its initial position to handle the next container. Once a TC receives a container, it needs 15 time units to finish the delivery to the yard. Since the QC is a heavy equipment, it is expected that the utilization of the QC is higher. We thus have the requirement that the accumulated time of the QC waiting for an idle TC is at most one twentieth of the time in any interval.

The automata QC and TC are shown in Fig 6. V2 is the location at which QC waits for an idle TC. The boolean variable `idle[i]` is used to indicate whether TC[`i`] is idle (it takes value 1 when idle) or not with an initial value 1. The urgent channel `down[i]`

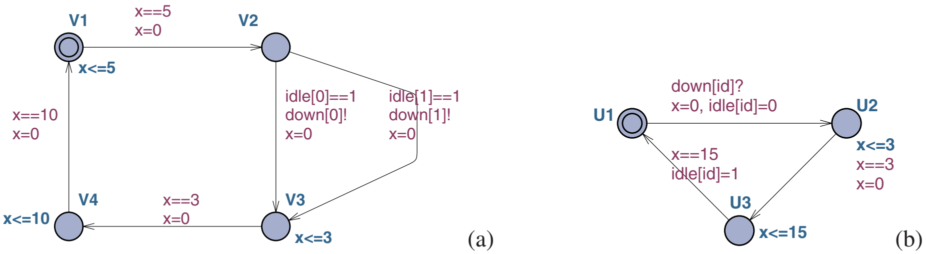


Fig. 6. Automated container system: (a) QC automaton (b)TC automaton

ensures that if the QC is at location V2, and as soon as TC[i] is idle, then the container unloading is done immediately. The whole system is $\mathcal{A}$ =QC$\|$ TC[0]$\|$ TC[1]. The above requirement can be easily specified by the following DC formula:

$$\ell > 0 \Rightarrow \textstyle\int(V2, \times_2, \times_3) \leq 0.05\ell \tag{4}$$

The above formula can be easily transformed to the following LDI:

$$D : \ell > 0 \Rightarrow 19\textstyle\int(V2, \times_2, \times_3) - \int(\times_1, \times_2, \times_3) \leq 0 \tag{5}$$

The *critical* location vector is $s_1 = (V2, \times_2, \times_3)$ and $Q = 2000$. We also declare `system` QC0,TC0,TC1 $< \mathcal{S}$. Following the techniques in Section 3, the transformed QC, the auxiliary automaton $\mathcal{S}$ and the function `accum`() are given in Fig.7 and Fig.8.

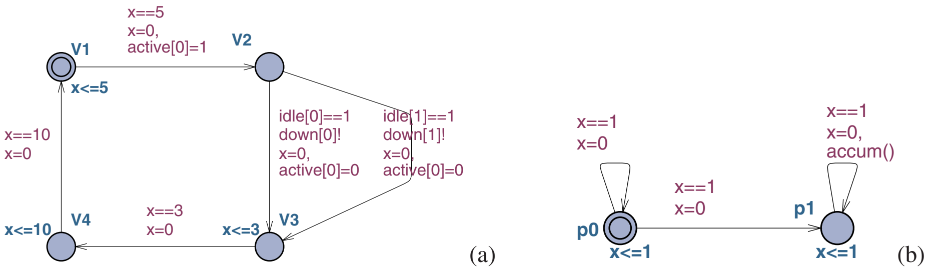


Fig. 7. Automated container system: (a) Transformed QC automaton (b) $\mathcal{S}$ automaton

```
void accum()
{      $gc = (gc < A?gc + 1 : A)$
    if $active[0] == 1$
        $d = (gc \geq A \wedge d > M?M + 1 : d + 19)$
    $d = (gc \geq A \wedge d < M - Q?d : d - 1)$
}
```

Fig. 8. Function `accum`()

The failure state is specified as C : `A[] not` $d > 0$. We checked whether the failure state is never reached with UPPAAL and got $\mathcal{G} \parallel \mathcal{S} \models C$. Therefore, we have $\mathcal{A} \models D$. If the unloading time from a TC to the yard is changed to a big value, e.g, 35, C does not hold any more and UPPAAL can generate a counter-example.

This case can also be extended to a more complicated system with more QCs and TCs. However, the transformed automata and $\mathcal{S}$ do not change if the LDI remains unchanged.

5 Conclusion

This paper studies the problem of automatic verification of a network of timed automaton against an LDI. To solve this problem, several algorithms have been proposed in the literature [3,10]. For improving the complexity the algorithm proposed in [11] is restricted to the class of the so-called *digitalized properties*. However, these model checking algorithms can only apply to one automaton, and cannot deal with the case when B is infinite in an efficient way. In addition, there is no available tool to support these algorithms. Recently some works have been done [17,18,19] for developing model checking tools for Duration Calculus. However, to our knowledge, compared with the model checkers of other temporal logics, the tools are still not widely applicable in industrial fields.

In [16], we give an algorithm to reduce model checking an LDI to model checking a CTL formula. The basic idea of that algorithm is: Instead of checking an automaton $\mathcal{A}$ against the LDI directly, we first construct an untimed model $\mathcal{H}$ from $\mathcal{A}$ and the LDI, and then construct a CTL formula ϕ from the LDI and then use a popular model checker, such as UPPAAL or SPIN, to check if $\mathcal{H} \models \phi$. This technique is simple and works well for one automaton. However, in order to apply the approach to real-time systems modelled as a network of automata with a common set of clocks and actions, we have to construct a composite automaton of the network explicitly. Obviously, it is not always feasible to manually construct such a composite automaton because of the high complexity and mistakes may be inevitable.

To avoid the construction of composite automata, in this paper, we have presented a different approach to this problem. We first construct a network of automata $\mathcal{G}$ to record the entry end exit of the *p-critical* locations. The construction of each $\mathcal{G}_i$ is very similar to the automaton $\mathcal{A}_i$ itself, and is simpler than the transformed model $\mathcal{H}$ proposed in [16]. As we need to check from any reachable state, when the antecedent of the LDI is true, whether or not the consequence is true, we then introduce $\mathcal{S}$ to count the observation time gc and the sum of the durations of the critical locations d from any reachable state. The trick that $\mathcal{S}$ can stay in the initial state for arbitrary time units ensures that $\mathcal{S}$ starts the calculation of gc and d from any reachable state of $\mathcal{G}$. Also, the introduction of $\mathcal{S}$ is convenient for the user to simplify the specification. Without this, more extra variables and channels need to be introduced in the transformed network $\mathcal{G}$ and more complex expression of temporal logic needs to be defined. Finally, we define a failure state such that $\mathcal{A} \models \mathcal{D}$ if and only if this failure state is never reached in $\mathcal{S}$. Such checking can be done by some popular model checkers like UPPAAL.

In our future work, we will implement a tool that integrates the construction of $\mathcal{S}$ and $\mathcal{G}$. This tool is able to transform a `xml` file that has been constructed in UPPAAL to describe the original automata $\mathcal{A}$ to two `xml` files that describe respectively transformed $\mathcal{S}$ and $\mathcal{G}$. Then UPPAAL uses these two files as the input to do the checking. In this way, the checking of an LDI can be done without manually constructing the transformed automata. These will help to make Duration Calculus more applicable in practical applications.

The other direction of the future work is to apply the technique and tool to schedulability analysis and scheduler synthesis. There have been other approaches to formalising real-time scheduling, for instance, in [20] TLA is used to specify a system and analyze the schedulability of the system by proving that the system and the scheduler satisfy the given scheduling constraint. Using the Duration Calculus, Zhou Chaochen et al formalized a well-established scheduler EDF [22] and defined the semantics of scheduled programs [21]. We believe that these techniques will be useful in our future work on real-time scheduling analysis and synthesis, based on model checking DC properties.

Acknowledgment. We are grateful to Anders P. Ravn for his comments on how to improve the presentation, and in particular for his suggestion on the algorithm of the calculation of the sum of the durations that simplified the algorithm in an earlier version of the paper. Without his help, this paper would not have become its present form.

References

1. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Alur, R.: Timed Automata. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 8–22. Springer, Heidelberg (1999)
3. Braberman, V.A., Van Hung, D.: On Checking Timed Automata for Linear Duration Invariants. In: Proc. RTSS 1998, pp. 264–273. IEEE Computer Society Press, Los Alamitos (1998)
4. Emerson, E.A., Halpern, J.Y.: “Sometimes” and “Not Never” Revisited: On Branching versus Linear Time Temporal Logic. Journal of the ACM 33(1), 151–178 (1986)
5. Henzinger, T., Manna, Z., Pnueli, A.: What Good Are Digital Clocks? In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992)
6. Chakravorty, G., Pandya, P.K.: Digitizing Interval Duration Logic. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 167–179. Springer, Heidelberg (2003)
7. Bosnacki, D.: Digitization of Timed Automata. In: Proc. FMICS 1999, pp. 283–302 (1999)
8. Van Hung, D., Giang, P.H.: Sampling Semantics of Duration Calculus. In: Jonsson, B., Parrow, J. (eds.) FTRTFT 1996. LNCS, vol. 1135, pp. 188–207. Springer, Heidelberg (1996)
9. Franzle, M.: Model-Checking Dense-Time Duration Calculus. Formal Asp. Comput. 16(2), 121–139 (2004)
10. Li, X., Van Hung, D.: Checking Linear Duration Invariants by Linear Programming. In: Jaffar, J., Yap, R.H.C. (eds.) ASIAN 1996. LNCS, vol. 1179, pp. 321–332. Springer, Heidelberg (1996)
11. Thai, P.H., Van Hung, D.: Verifying Linear Duration Constraints of Timed Automata. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 295–309. Springer, Heidelberg (2005)
12. Chaochen, Z., Hoare, C.A.R., Ravn, A.P.: A Calculus of Durations. Information Processing Letters 40(5), 269–276 (1991)
13. Zhou, C., Zhang, J., Yang, L., Li, X.: Linear Duration Invariants. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994 and ProCoS 1994. LNCS, vol. 863, pp. 86–109. Springer, Heidelberg (1994)
14. Zhou, C., Hansen, M.R.: Duration Calculus. A Formal Approach to Real-Time Systems (2004)
15. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
16. Zhang, M., Van Hung, D., Liu, Z.: Verification of Linear Duration Invariants by Model Checking CTL Properties. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 395–409. Springer, Heidelberg (2008)

17. Pandya, P.K.: Interval Duration Logic: Expressiveness and Decidability. ENTCS 65(6) (2002)
18. Meyer, R., Faber, J., Rybalchenko, A.: Model Checking Duration Calculus: A Practical Approach. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) ICTAC 2006. LNCS, vol. 4281, pp. 332–346. Springer, Heidelberg (2006)
19. Fränzle, M., Hansen, M.R.: Deciding an Interval Logic with Accumulated Durations. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 201–215. Springer, Heidelberg (2007)
20. Liu, Z., Joseph, M.: Specification and Verification of Fault-Tolerance, Timing, and Scheduling. ACM Trans. Program. Lang. Syst. 21(1), 46–89 (1999)
21. Zhou, C., Hansen, M.R., Ravn, A.P., Rischel, H.: Duration Specifications for Shared Processors. In: Vytopil, J. (ed.) FTRTFT 1992. LNCS, vol. 571, pp. 21–32. Springer, Heidelberg (1991)
22. Zheng, Y., Zhou, C.: A Formal Proof of the Deadline Driven Scheduler. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994 and ProCoS 1994. LNCS, vol. 863, pp. 756–775. Springer, Heidelberg (1994)

Automata Based Model Checking for Reo Connectors

Marcello M. Bonsangue[1,2] and Mohammad Izadi[1,3,4]

[1] LIACS - Leiden University, The Netherlands
[2] Centrum voor Wiskunde en Informatica (CWI), The Netherlands
[3] Dept. of Computer Engineering, Sharif University of Technology, Tehran, Iran
[4] Research Institute for Humanities and Cultural Studies, Tehran, Iran

Abstract. Reo is a connector language for the exogenous composition and orchestration of components in a software system. An operational semantics of Reo connectors can be given in terms of Büchi automata over a suitable alphabet of records, capturing both synchronization and context dependency requirements. In this paper, we define an action based linear time temporal logic for expressing properties of Reo connectors. Formulas can be synthesized into Büchi automata representing Reo connectors, thus leading to an automata based model checking algorithm. By generalizing standard automata based model checking algorithms for linear time temporal logic, we give an efficient on-the-fly algorithm for the model checking of formulas for Reo connectors.

1 Introduction

Reo [1] is a coordination language based on connectors for the orchestration of components in component based systems. Primitive connectors such as synchronous channels or FIFO queues are composed to build circuit-like component connectors which exhibit complex behavior and play the role of glue code in exogenously coordinating the components to produce a system.

Reo generalizes dataflow networks and Khan networks because it allows to express behavior including state-based, context dependent, multi-party synchronization and mutual exclusion. Typically, the operational description of Reo connectors is given in terms of constraint automata [4]. Constraint automata are acceptors of timed data streams (also known as abstract behavioral types) [2], but are much more concrete and suitable for model checking analysis. A constraint automaton is a labeled transition system in which each transition label contains two parts: a set N of port names that are *synchronized* if the transition is taken and a proposition g on the data. The latter acts as constraint on the data that could be communicated through the ports in N. The data flowing through the ports in N is *mutually exclusive* with respect to any communication by ports not in N.

Two specific shortcomings of the constraint automata model of Reo, are that it cannot model desired *fairness* constraints and that it cannot model connectors with behavior depending upon the absence of communication requests on a connector boundary. This latter feature is called *context dependency* [7].

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 260–275, 2010.

In [12] we have shown that every constraint automaton can be translated into an essentially equivalent Büchi automaton. The basic idea is the use of records as data structures for modeling *multi-party synchronization* as well as *mutual exclusion*: ports in the domain of a record are allowed to communicate simultaneously the data assigned to them, while ports not in the domain of the record are blocked so that no communication can happen. Because our model is based on Büchi automata, we can easily express *fairness conditions* admitting only executions for which some actions occur infinitely many times [21].

In order to address context-dependent behavior, we extended in [13] our model with the possibility of testing if some ports of the environment are willing to communicate or not. More concretely, we considered a Büchi variant of Kozen's finite automata on guarded strings [16]. An *infinite guarded string* is an alternating sequence of sets of *ready* ports and records of *fired* ports (together with their respective data flow). This idea has been similarly and independently pursued for another extension of constrain automata, called intensional automata [8].

In this paper, we continue our line of work by presenting a variant of linear time temporal logic that is interpreted over *infinite guarded strings.* Traditional temporal logics [5] ignore the actions of the system specified, and concentrate on the sequence of states a system goes through. Inspired by an extension of linear time temporal logic to include modalities to talk about actions [18], we use propositions for the specification of sets of ports that must be ready to communicate at the connector boundary, and next-state modalities indexed by records for the specification of the one-step data flow in the connector.

The logic is very simple and yet powerful enough to specify mutual exclusion, multi-party synchronization, context dependencies and fairness constraints. We show through some examples how our logic can serve as a specification formalism for Reo connectors. The main results of this paper are an adaption of the construction of a Büchi automaton from a temporal formula [22] to our logic and the presentation of an efficient on-the-fly algorithm for the model checking of our logic for Reo connectors.

Forgetting about time, our logic has a similar goal of the *time scheduled-data-stream logic* (TSDSL) [3]. An interesting feature of TSDSL is the combination of standard LTL operators with path modality that allow to reason about the timed data streams by means of regular expressions. In this paper, we show that the untimed fragment of TSDSL can be embedded in our logic even if the latter does not have path modalities. The inclusion is in fact a consequence of the fact that (extended) regular expressions can be encoded in a linear time temporal logic with actions. However our logic is more expressive than TSDSL, as it allows for example to distinguish between non-deterministic and context dependent behavior. Furthermore, only a rough sketch for model checking a TSDSL has been given in [3], which combines the standard LTL model checking approach [21] with automata synthesis from regular expressions [11].

A branching time-abstract variant of TSDSL, called BTSL, is introduced in [15]. It is a minor variation of a fragment of the *reconfiguration logic* previously introduced in [6], tailored towards an efficient model checking. While in our automata-based approach the state-space is generated dynamically and only the minimal amount of information required by the verification procedure is stored in memory, the approach to model checking in [15] is symbolic, where the entire constraint automaton for a Reo connector is represented by means of a binary decision diagram [5].

The remainder of this paper is organized as follows. In Section 2 we present an informal introduction to the Reo connectors. A formal model of Reo connectors in terms of Büchi automata is given in Section 3. In Section 4 we present a logic for specifying Reo connectors, and a global model checking algorithm is given in Section 5. We conclude with Section 6 where we give an efficient on the fly model checking algorithm.

2 Reo Connectors

Reo is an exogenous coordination language which is based on a calculus of component connectors [1]. In Reo software components are independent processes which communicate solely through ports. Ports are related by a network of connectors that specifies the glue code. These connectors build together what is called a coordination system.

A Reo connector consists of a set of source, sink and internal nodes, and a user-defined semantics. A connector may accept the data offered at source nodes by components, or produce data for sink nodes. Component coordination is achieved by delaying or synchronizing those operations. Reo connectors are composed by conjoining some of their source or sink nodes to form internal nodes not accessible from the environment.

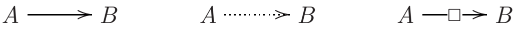


Fig. 1. Basic Reo connectors: synchronous, lossy synchronous, and FIFO1

Figure 1 shows the graphical representation of some binary Reo connectors whose composition allows for expressing a rich set of coordination strategies [1].

The *synchronous channel* is a connector with one source and one sink end. It accepts a data item through its source end if and only if it can simultaneously dispense it through its sink.

A *lossy synchronous channel* is similar to a synchronous channel, but it never delays the port at the source end. If the port at its sink is pending the channel transfers the data item, otherwise the data item is lost. The behavior of this channel is context-sensitive: it has to be able to sense the absence of a request for data from the port connected at the sink.

A *FIFO1 channel* is an asynchronous connector. Data from the source is accepted as long as the buffer is empty. The data item received is stored in the channel and communicated to the port at the sink node, when requested. More general FIFO channels with two or more buffer cells can be produced by composing several FIFO1 channels [4].

3 Reo and Büchi Automata

In this section we introduce an operational model of Reo based on Büchi automata. The idea is to model a Reo connector by an ordinary Büchi automaton with transitions labeled by *records* assigning port names to data. Having records as actions of our model reflects the assumption of Reo that only the data flow among the *synchronizing* ports of the connectors is observable. Since Reo connectors are reactive, we describe their behavior in terms of all their infinite executions, i.e. as the set of all possible infinite runs accepted by the automaton.

3.1 Records for Synchronization

To begin with, let us first recall the concept of record. Let $\mathcal{N}$ be a finite nonempty set of port names and $\mathcal{D}$ be a finite nonempty set of data. We write $Rec_{\mathcal{N}}(\mathcal{D}) = \mathcal{N} \rightharpoonup \mathcal{D}$ for the set of *records* with entries from a set of data $\mathcal{D}$ and labels from a set of port names $\mathcal{N}$, consisting of all *partial* functions from $\mathcal{N}$ to $\mathcal{D}$. For a record $r \in Rec_{\mathcal{N}}(\mathcal{D})$ we write $dom(r)$ for the domain of r. Sometimes we use the more explicit notation $r \doteq [n_1 = d_1, \ldots, n_k = d_k]$ for a record $r \in Rec_{\mathcal{N}}(\mathcal{D})$, with $dom(r) = \{n_1, \ldots, n_k\}$ and $r(n_i) = d_i$ for $1 \leq i \leq k$. We denote by τ the record with empty domain, that is $dom(\tau) = \emptyset$, and call it the *empty record.*

We use records as data structures for modeling constrained synchronization of ports in $\mathcal{N}$. Following [19], we see a record $r \in Rec_{\mathcal{N}}(\mathcal{D})$ as carrying both positive and negative information: only the ports in the domain of r have the possibility to exchange the data assigned to them by r, while the other ports in $\mathcal{N} \setminus dom(r)$ are definitely constrained to *not* perform any communication. This intuition is formalized by the fact that only for ports $n \in dom(r)$ data can be retrieved, using *record selection* $r.n$. Formally, $r.n$ is just (partial) function application $r(n)$.

For simplicity, in this paper we do not distinguish between input and output communications. In fact, a record merely reports the data value exchanged at a port, but not whether it has been received or sent. Clearly, such a distinction can be incorporated by using either a set of port names or a data domain that distinguishes between these two types of values.

3.2 Büchi Automata of Records

A stream (i.e., an infinite string) of records describes a possible data flow among the ports of a system. Sets of streams of records are just languages, and as such some of them can be recognized by ordinary Büchi automata. Next we recall some basic definitions and facts on Büchi automata [23].

Definition 1. *A* Büchi automaton *is a tuple $B = \langle Q, \Sigma, \longrightarrow, Q_0, F\rangle$ where, Q is a finite set of* states*, Σ is a finite nonempty set of symbols called* alphabet*, $\longrightarrow \subseteq (Q \times \Sigma \times Q)$ is a* transition relation*, $Q_0 \subseteq Q$ is a nonempty set of* initial *states and $F \subseteq Q$ is a set of* accepting (final) *states.*

We often write $q \xrightarrow{a} p$ instead of $(q, a, p) \in \longrightarrow$. For technical convenience, we view a terminating computation as an execution repeating for ever its last state.

A model of a (finite state) Reo connector over a finite set of port names $\mathcal{N}$ and finite set of data $\mathcal{D}$ consists of a Büchi automaton $B = \langle Q, \Sigma, \longrightarrow, Q_0, F\rangle$ where $\Sigma = Rec_{\mathcal{N}}(\mathcal{D})$ together with a valuation function $V : Q \to 2^{2^{\mathcal{N}}}$, assigning to each state in Q truth values to each subset of ports in $\mathcal{N}$ (or, equivalently, assigning to each state a Boolean expression over ports in N). The intuition is that names in $\mathcal{N}$ are the port at the boundary of a connector, Q describes the states of the connector, and a transition $q \xrightarrow{r} p$ gives a possible one-step behavior, meaning that each port $A \in dom(r)$ has the possibility to receive or send the data item $r.A$ leading from a state q to a state p, while the other ports in $\mathcal{N} \setminus dom(r)$ do not perform any data communication. Finally the valuation function V tells us the possible sets of ports of the connector that must be ready to communicate in a given state before an outgoing transition is taken from that state.

We refer to such a model as a *Büchi automaton (on streams) of records* abbreviated by BAR. The model of Reo connector here differs from that of [13] so as to simplify the presentation of the model checking algorithm to be introduced in Section 5, but the difference is inessential. In [13] the states of an automaton modeling a Reo connector are labeled by Boolean expressions over the set of port names $\mathcal{N}$, whereas here we considered $2^{2^{\mathcal{N}}}$. Clearly, the two structures are isomorphic up to Boolean equivalences.

The BARs depicted in Figure 2 are models of some of the basic Reo connectors between two ports A and B over a singleton data set $\{d\}$. Pictorially, we label states with sets of ports which evaluated to true. The leftmost BAR is a model a synchronous channel: it accepts a data item through one of its ports if and only if it can simultaneously dispense it through its other port. Both ports A and B must be willing to communicate before the transition is taken.

The rightmost BAR is a model of a FIFO1 connector initially empty. In that state, data from the port A is accepted independently from the port B willingness to communicate, which explain the two sets of ports A and AB. The data item received is stored and eventually communicated to the port B, when B is enabled (independently from A willing to start a new communication).

The BAR in the middle is more interesting. It is a model of the lossy synchronous channel. Data items from port A are always accepted, and transferred synchronously to the port B if B is ready to accept, otherwise the data item is lost. The context sensitive behavior of this connector is obtained by acting differently depending on the presence or not of the port B in the set of ports ready to communicate.

BARs are acceptors of infinite guarded strings [14] over the alphabet $Rec_{\mathcal{N}}(\mathcal{D})$. An *infinite guarded string* is an alternating infinite sequence $N_0 r_0 N_1 r_1 \cdots$ where $r_i \in Rec_{\mathcal{N}}(\mathcal{D})$ and $dom(r_i) \subseteq N_i$ for all $i \geq 0$. Intuitively, a guarded string represents an execution of a Reo connector, where for each step it records the ports ready for a communication and the actual data flow among a subset of them. The constraint $dom(r_i) \subseteq N_i$ is in line with the intuition that if some ports

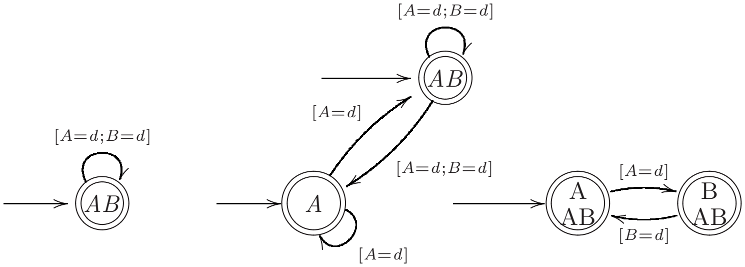


Fig. 2. BAR models of basic Reo connectors

can take part in a communication, then all those ports (and possibly more) must be ready to communicate (enabled).

Definition 2. *An* infinite computation *for a guarded string $N_0 r_0 N_1 r_1 \cdots$ in a BAR is an infinite sequence $q_0, r_0, q_1, r_1, ...,$ of alternating states and records in which $q_0 \in Q_0$, $V(q_i)(N_i)$ is true and $q_i \xrightarrow{r_i} q_{i+1}$ for all $i \in \mathbb{N}$. An infinite guarded string γ is* accepted *by a BAR B if there is an infinite computation for γ in B with at least one of the final states occurring infinitely often.*

The *language* of a BAR B, denoted by $L(B)$, is the set of all infinite guarded strings accepted by it.

We say that two BARs B_1 and B_2 are *language equivalent* if $L(B_1) = L(B_2)$. For example, we can delete from a BAR B all its *inconsistent states* obtaining a language equivalent BAR, where a state is said to be inconsistent if every set of ports (including the empty one) is evaluated to false. Also, we can delete from a BAR B all its *inconsistent transitions* obtaining a language equivalent BAR, where a transition $q \xrightarrow{r_i} p$ is inconsistent if $V(q)(S)$ is false for all $S \supseteq dom(r_i)$.

Because BARs are based on ordinary Büchi automata, several kinds of fairness requirements on Reo connectors can be easily modeled. We just give an example here. Consider a *merger* connecting two input ports A and B to a single output port C. It transmits synchronously data items from either A or B to the port C only if all three ports are wiling to communicate. In this case, only one of the two input ports is chosen non-deterministically. Two BAR model of this merger connector are shown in Figure 3. The rightmost model allows unfair executions in which data values from the same source are always preferred. The leftmost BAR model disallows those unfair executions.

We conclude this section by mentioning that using the richer structure of the alphabet of BARs, one can give a general definition of synchronous product that works even if the alphabets of the two automata are different. The idea is that two transitions synchronize if they are labeled by compatible records (i.e. on the common ports they communicate the same data values), whereas they are interleaved if they are labeled with records not referring to ports of the other automaton. Valuations of sets of names are conjoined after being extended to the ports of the other automaton. More details on the join operator of two BARs as well as on the hiding of ports can be found in [12,13].

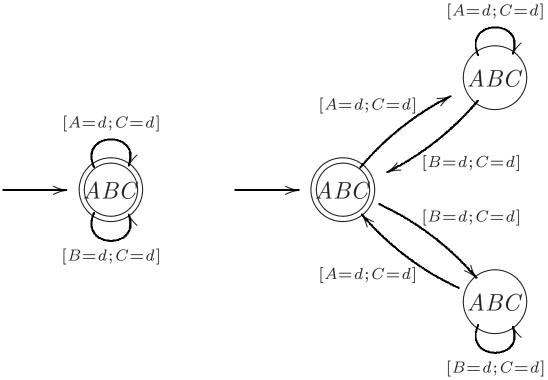


Fig. 3. Merger connectors

4 Record-Based Linear-Time Temporal Logic

In this section we introduce a record based linear time temporal logic (ρLTL) which is an extension of linear time temporal logic (LTL) [21] for reasoning about data flow, synchronization and context dependencies of Reo connectors. We use as atomic propositions sets of port names, indicating the ports ready to communicate, and index the usual next state operator of LTL with a record, for the specification of communicating ports and of their respective data flow.

Definition 3. *The set of* ρLTL formulas *over a finite set of port names $\mathcal{N}$ and finite set of data $\mathcal{D}$ is defined inductively by the following syntax:*

$$\phi ::= \; N \,|\, \neg\phi \,|\, \phi \vee \phi |\, \langle r\rangle\phi \,|\, \phi U \phi.$$

where $N \subseteq \mathcal{N}$ and $r \in Rec_{\mathcal{N}}(\mathcal{D})$.

Formulas of ρLTL are interpreted over infinite guarded strings. A necessary condition to interpret a formula for a guarded string is that both use the same set of port names $\mathcal{N}$ and data set $\mathcal{D}$, which will be assumed in the sequel. Intuitively, N holds for a guarded string if N is the first *guard* of the string, whereas $\langle r\rangle\phi$ holds if r is the first *action* of the string and ϕ holds for its remaining suffix.

Formally, given an infinite guarded string $M = N_0 r_0 N_1 r_1 \cdots$, we define M^i as the suffix $N_i, r_i, N_{i+1}, r_{i+1}, \cdots$, for $i \geq 0$. Recall that we consider only guarded strings for which $r_i \in Rec_{\mathcal{N}}(\mathcal{D})$ and $dom(r_i) \subseteq N_i$ for all $i \geq 0$. The semantics of a ρLTL formula is defined inductively as follows:

$$\begin{array}{ll} M \models N & \text{iff } N_0 = N \\ M \models \phi_1 \vee \phi_2 & \text{iff } M \models \phi_1 \text{ or } M \models \phi_2 \\ M \models \neg\phi & \text{iff } M \not\models \phi \\ M \models \langle r\rangle\phi & \text{iff } r_0 = r \text{ and } M^1 \models \phi \\ M \models \phi_1 U \phi_2 & \text{iff } \exists j \geq 0 \text{ such that} M^j \models \phi_2 \text{ and } \forall 0 \leq i < j, M^i \models \phi_1 \end{array}$$

As usual, we denote by $\| \phi \|$, the set of all models of the ρLTL formula ϕ, and define logical equivalence between ρLTL formulas as $\phi_1 \equiv \phi_2$ if and only if $\| \phi_1 \| = \| \phi_2 \|$. If B is a BAR and ϕ a ρLTL formula, we say that $B \models \phi$ if $L(B) \subseteq \| \phi \|$.

Several other operators can be derived from the basic operators of ρLTL. The Boolean operators $\wedge$ and $\rightarrow$ are derived in the obvious way. We can write *true* for instance as $N \vee \neg N$, for some $N \subseteq \mathcal{N}$. The temporal modalities *eventually* and *always* can be derived as usual, by setting $\Diamond\phi = \mathit{true} U \phi$ and $\Box\phi = \neg\Diamond\neg\phi$, respectively. The dual operator of the *until* is the *release* operator defined by $\phi\, R\, \psi = \neg(\neg\phi\; U\; \neg\psi)$. The *weak* variant '$W$' of the until operator is obtained as $\phi\; W\; \psi = (\phi\; U\; \psi) \vee \Box\phi$. The dual operator of $\langle r\rangle\phi$ is $[r]\phi = \neg\langle r\rangle\neg\phi$, which intuitively holds for a guarded string if either its first action is different from r or its continuation satisfies ϕ. In fact, $[r]\phi \equiv \neg\langle r\rangle \mathit{true} \vee \langle r\rangle\phi$. For example, the formula $[r]\mathit{false}$ is satisfied by all guarded strings with as first action a record different from r.

4.1 Some Useful Encodings

The standard *next* operator of linear time temporal logic is defined as $\bigcirc\phi = \bigvee_{r \in Rec_{\mathcal{N}}(\mathcal{D})} \langle r\rangle\phi$. It is not hard to see that the next operator is self-dual, in the sense that $\neg \bigcirc \phi \equiv \bigcirc\neg\phi$. Further, because our models are infinite strings, $\bigcirc \mathit{true} \equiv \mathit{true}$, meaning that connectors are reactive and cannot stop the data flow (progress is always possible).

A *data constraint* δ for a set of names $N \subseteq \mathcal{N}$ is a satisfiable propositional formula built from the atoms '$d_A \in P$', '$d_A = d$', and '$d_A = d_B$', where $A, B \in N$, $d \in \mathcal{D}$ and $P \subseteq \mathcal{D}$. Data constraints, together with a set of names on which they are defined can be viewed as a symbolic representation of a set of records. We can therefore define a derived operator $\langle N, \delta\rangle\phi$, for δ a data constraint for N, by setting

$$\langle N, \delta\rangle\phi = \bigvee\{\langle r\rangle\phi | dom(r) = N, r \models \delta\},$$

where $r \models (d_A \in P)$ if $r.A \in P$, $r \models (d_A = d)$ if $r.A = d$ and $r \models (d_A = d_B)$ if $r.A = r.B$ (disjunction and negation are as expected).

In [3], timed scheduled-data expressions are introduced to specify data stream logic. Leaving out time, *scheduled-data expressions* are ordinary regular expressions built from either data constraints or records. Scheduled-data expressions α are incorporated in data stream logic of [3] by formulas of the type $\langle\langle\alpha\rangle\rangle\phi$. We can encode them in ρLTL by using the following inductive translation:

$$\begin{array}{llll}
\langle\langle 0\rangle\rangle\phi & = \mathit{false} & \langle\langle 1\rangle\rangle\phi & = \phi \\
\langle\langle N, \delta\rangle\rangle\phi & = \langle N, \delta\rangle\phi & \langle\langle r\rangle\rangle\phi & = \langle r\rangle\phi \\
\langle\langle \alpha_1 + \alpha_2\rangle\rangle\phi & = \langle\langle\alpha_1\rangle\rangle\phi \vee \langle\langle\alpha_2\rangle\rangle\phi & \langle\langle \alpha_1 \times \alpha_2\rangle\rangle\phi & = \langle\langle\alpha_1\rangle\rangle\phi \wedge \langle\langle\alpha_2\rangle\rangle\phi \\
\langle\langle \alpha_1;\alpha_2\rangle\rangle\phi & = \langle\langle\alpha_1\rangle\rangle(\langle\langle\alpha_2\rangle\rangle\phi) & \langle\langle \alpha^*\rangle\rangle\phi & = \langle\langle\alpha\rangle\rangle \mathit{true} U \phi
\end{array}$$

where 0 is the unit with respect to the union operator +, and 1 is the unit with respect to the composition operator ;. In fact we have

$$\langle\langle 0 + \alpha\rangle\rangle\phi \equiv \langle\langle 0\rangle\rangle\phi \vee \langle\langle\alpha\rangle\rangle\phi \equiv \mathit{false} \vee \langle\langle\alpha\rangle\rangle\phi \equiv \langle\langle\alpha\rangle\rangle\phi$$
$$\langle\langle 1;\alpha\rangle\rangle\phi \equiv \langle\langle 1\rangle\rangle(\langle\langle\alpha\rangle\rangle\phi) \equiv \langle\langle\alpha\rangle\rangle\phi\,.$$

Scheduled-data expressions allow to express formulas that hold only for externally observable steps, thus not sensible to a finite number of internal steps. Given a ρLTL formula ϕ, we define $\Diamond_\tau\phi = \langle\langle\tau^*\rangle\rangle([\tau]\mathit{false} \wedge \phi)$. Informally, $\Diamond_\tau\phi$ holds if ϕ holds after finitely many internal τ steps.

4.2 Specifying Reo Connectors

Next we present few examples of specifications of basic Reo connectors. First of all, let us consider a synchronous channel from a port A to a port B. If both ports are enabled, then the channel must let the data flow. This can be expressed by the following ρLTL formula:

$$\Box(\{A, B\} \to \langle\langle AB, d_A = d_B\rangle\rangle \mathit{true})\,. \tag{1}$$

The above formula is clearly satisfied by our BAR model of synchronous channel in Figure 2. However, it is also satisfied by the BAR modeling a lossy synchronous channel. This because (1) does not guarantee that data cannot flow through a single port. This is done by adding to the specification of a synchronous channel the following

$$\Box(\neg\langle\langle A, \mathit{true}\rangle\rangle \mathit{true} \vee \neg\langle\langle B, \mathit{true}\rangle\rangle \mathit{true})\,.$$

The above formula does not hold for the lossy synchronous channel. In fact, for such a connector it holds that if the port A is enabled but B is not, then the data at A is lost. This is expressed by

$$\Box((\{A\} \wedge \neg\{A, B\}) \to \langle\langle A, \mathit{true}\rangle\rangle \mathit{true})$$

Further, in a lossy synchronous channel, data cannot flow through port B alone, that is $\Box\neg\langle\langle B, \mathit{true}\rangle\rangle \mathit{true}$.

Differently from the two previous channels, a FIFO1 channel is asynchronous, meaning that data does not flow simultaneously through its ports A and B, that is $\Box\neg\langle\langle AB, \mathit{true}\rangle\rangle \mathit{true}$. Further, a data item received through port A is never lost, as it is output to port B as soon as B is enabled. Of course, this does not need to be immediate and it can even be the case that B is never enabled. This is specified by means of a weak until operator allowing possibly infinitely many internal steps between the two observable actions:

$$\Box \bigwedge_{d\in\mathcal{D}} \langle[A = d]\rangle(\langle\tau\rangle \mathit{true} \wedge \neg(\{B\} \vee \{A, B\}))\ W\ \langle[B = d]\rangle)\,.$$

To complete the specification of a FIFO1 channel, we need the converse of the above property, saying that after data flows through port B the store of the channel is empty and hence a new data item can flow through port A as soon as A is enabled:

$$\Box\langle\langle B, \mathit{true}\rangle\rangle(\langle\tau\rangle \mathit{true} \wedge \neg(\{A\} \vee \{A, B\}))\ W\ \langle\langle A, \mathit{true}\rangle\rangle)\,.$$

Thus in a FIFO1 channel, data flows through two ports alternately, and never simultaneously.

5 From Formulas to Automata: Model Checking

Next we introduce a global translation of ρLTL formulas into BARs. Our construction is based on the translation from ordinary LTL formulas to Büchi automata [22], adapted so to take into account the next state operator indexed by records. For simplicity, the resulting BAR will have multiple sets of acceptance states in which, a run is accepted if and only if for each acceptance set there exists at least one state that appears infinitely often in that run. To obtain an ordinary BAR, one can use the fact that for each generalized Büchi automaton there is a language equivalent ordinary Büchi automaton [23].

We begin by defining the closure $CL(\phi)$ of a ρLTL formula ϕ as the set $CL(\phi) = CL'(\phi) \cup \{\neg\psi | \psi \in CL'(\phi)\}$, where we identify $\neg\neg\phi$ with ϕ and $CL'(\phi)$ is defined as the smallest set such that

- $\phi \in CL'(\phi)$,
- if $\neg\psi \in CL'(\phi)$ then $\psi \in CL'(\phi)$,
- if $\phi_1 \vee \phi_2 \in CL'(\phi)$ then $\phi_1, \phi_2 \in CL'(\phi)$,
- if $\langle r\rangle\psi \in CL'(\phi)$ then $\psi \in CL'(\phi)$,
- if $\neg\langle r\rangle\psi \in CL'(\phi)$ then $\neg\langle r\rangle \mathit{true}, \langle r\rangle\neg\psi \in CL'(\phi)$,
- if $\phi_1 \; U \; \phi_2 \in CL'(\phi)$ then $\phi_1, \phi_2, \langle r\rangle(\phi_1 \; U \; \phi_2) \in CL'(\phi)$ for all $r \in \mathit{Rec}_{\mathcal{N}}(\mathcal{D})$.

The set $CL(\phi)$ is finite, and its size is linear in the size of the formula ϕ.

The states of the BAR associated with a formula ϕ are the propositionally and temporally consistent subsets of $CL(\phi)$, the so called *atoms*. Formally, an atom $A \subseteq CL(\phi)$ is a set such that

1. $\psi \in A$ if and only if $\neg\psi \notin A$,
2. $\phi_1 \vee \phi_2 \in A$ if and only if $\phi_1 \in A$ or $\phi_2 \in A$,
3. $\neg\langle r\rangle\phi \in A$ if and only if $\neg\langle r\rangle \mathit{true} \in A$ or $\langle r\rangle\neg\phi \in A$,
4. $\phi_1 \; U \; \phi_2 \in A$ if and only if $\phi_1 \in A$ or $\phi_2, \langle r\rangle(\phi_1 \; U \; \phi_2) \in CL'(\phi)$ for some $r \in \mathit{Rec}_{\mathcal{N}}(\mathcal{D})$.

Note that latter two items can be explained by the following equivalences

$$\neg\langle r\rangle\phi \equiv [r]\neg\phi \equiv \neg\langle r\rangle \mathit{true} \vee \langle r\rangle\neg\phi \quad \text{and} \quad \phi_1 U \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \bigcirc(\phi_1 U \phi_2))$$

already discussed at the end of Section 4.

Definition 4. *Let ϕ be an ρLTL formula over names $\mathcal{N}$ and data $\mathcal{D}$. We define $BAR(\phi) = \langle Q, \mathit{Rec}_{\mathcal{N}}(\mathcal{D}), \rightarrow, Q_0, \mathcal{F}, V\rangle$ to be the generalized Büchi automaton of records such that*

- *Q is the set of atoms of ϕ,*
- *Q_0 is the set of atoms containing ϕ itself,*
- *$V(q)(N) = \mathit{true}$ if and only if $N \in q$, for every $N \subseteq \mathcal{N}$,*
- *$q \xrightarrow{r} p$ if and only if $\langle r\rangle\phi \in q$, $\phi \in p$, and $r \neq r'$ for all $\neg\langle r'\rangle \mathit{true} \in q$.*
- *$\mathcal{F}$ consists of the accepting sets $F_{\alpha U \beta} = \{q \in Q | \alpha \; U \; \beta \notin q \; or \beta \in q\}$ for each $\alpha U \beta \in CL(\phi)$.*

The following theorem shows the correctness of the above construction:

Theorem 1. *Let ϕ be a ρLTL formula over names $\mathcal{N}$ and data $\mathcal{D}$. The language accepted by $BAR(\phi)$ is the set of all models of ϕ, that is $L(BAR(\phi)) = \parallel \phi \parallel$.*

The above result can be used for an automata based procedure for model checking Reo connectors. Given a BAR model B of a Reo connector, and a ρLTL formula ϕ over the same set of port names $\mathcal{N}$ and data set $\mathcal{D}$, saying that $B \models \phi$ is equivalent to check whether $L(B)$ does not contain any models of $\neg\phi$. From the above theorem, this is equivalent to check if $L(B) \cap L(BAR(\neg\phi)) = \emptyset$. Therefore, if this intersection is empty, it proves that the connector B satisfies the property ϕ. Otherwise, every element of this intersection is a counterexample. Recall that intersecting two Büchi automata is just a simple extension of the product construction, and checking for emptiness is decidable [23]. The complexity of the model checking procedure is linear in the number of states of B and exponential in the length of the formula ϕ [21].

6 On-the-Fly Model Checking

In this section, we sketch an algorithm to construct the BAR for a ρLTL on-the-fly by generating the state space of the automaton incrementally, when required by the model checking procedure. The algorithm is a generalization of the on-the-fly approach proposed in [9] for standard LTL and extended with modalities for actions in a similar way as in [18].

For technical convenience we will work with ρLTL+ formulas, that is ρLTL formulas in positive forms:

$$\phi ::= \; N \,|\, \neg N \,|\, \phi \wedge \phi \,|\, \phi \vee \phi \,|\, \bigcirc \phi \,|\, \langle r \rangle \phi \,|\, [r]\phi \,|\, \phi U \phi \,|\, \phi R \phi$$

where $N \subseteq \mathcal{N}$ and $r \in Rec_{\mathcal{N}}(\mathcal{D})$. It is obvious that every ρLTL formula is equivalent to a positive one by pushing the negation inside every operator. The inclusion of the ordinary next state operator $\bigcirc\phi$ is to simplify the presentation.

The algorithm works by building a graph underlying the BAR to be defined for a formula ϕ. The nodes are labeled by sets of formulas which are obtained by decomposing into their sub-formulas according to their boolean structures. Temporal formulas are handled by just deciding what should be true at the node and what must be true at any next node. For an on-the-fly construction of the graph, we need to store some information at every node of the graph. More specifically, a node is a record containing the following fields:

1. *Incoming.* A set of elements of the form (q, X) where q is a node and $X \subseteq Rec_{\mathcal{N}}(\mathcal{D})$. Intuitively, a pair $(q, X) \in$ *Incoming* represents a transition from q to the current node labeled by the record r, for $r \in X$. A special element *init* is used to mark initial nodes.
2. *Old.* A set of formulas that have already been processed and hold in the current node (provided the properties in *New* are satisfied).

3. *New*. A set of formulas that have not yet been processed and that have to be satisfied in the current node.
4. *Next+*. A set of next-state formulas which this node satisfies. Thus they assert formulas that must be satisfied in any successor node.
5. *Next-*. A set of records which are *not* allowed to label outgoing transition from the current node.

The algorithm for building the graph of the automaton satisfying a ρLTL+ formula ϕ stores the nodes of the graph already computed in the list *Nodes_Set*. For all nodes in this list, it holds that the *New* field is empty. In this case, *Old* contains the set of formulas which the node satisfies. The full graph can then be constructed using the information in the *Incoming* field of each node.

The algorithm starts with a node q_0 with its *New* field set to $\{\phi\}$, $\mathit{Incoming} = \{\mathit{init}\}$ and with all other fields initially set to empty. When processing a node q the algorithm removes a formula ψ from its *New* field and tries all possible ways to satisfy it, by looking at the syntactic structure of ψ:

- If ψ is a (negation of a) subset of $\mathcal{N}$ then if $\neg\psi$ is in *Old* the node q is discarded because it contains a contradiction. Otherwise ψ is added to *Old*.
- If $\psi = \psi_1 \wedge \psi_2$ then both ψ_1 and ψ_2 are added to *New* because they both need to be satisfied in the node q.
- If $\psi = \psi_1 \vee \psi_2$ then a new node is created with the same fields as the current node q. Then ψ_1 is added to the *New* field of one node and ψ_2 to the other. The two nodes correspond to the two ways ψ can be satisfied.
- If $\psi = \bigcirc\varphi$ or $\psi = \langle r\rangle\varphi$ then ψ is added to the *Next+* field of the current node.
- The case when $\psi = [r]\varphi$ is novel with respect to the algorithm in [9]. Because $\psi \equiv \neg\langle r\rangle \mathit{true} \vee \langle r\rangle\varphi$, a new node is created with the same fields as the current node. The record r is added to the field *Next-* of one node, whereas the formula $\langle r\rangle\phi$ is added to the *Next+* field of the other node.
- If $\psi = \psi_1 \, U \psi_2$ then a new node is created with the same fields as the current node q. Because $\psi \equiv \psi_2 \vee (\psi_1 \wedge \bigcirc\psi)$, the formula ψ_2 is added to the *New* field of one node, while ψ_1 and $\bigcirc\psi$ are added to the fields *New* and *Next+* of the other node, respectively.
- If $\psi = \psi_1 R \psi_2$ then a new node is created with the same fields as the current node q. Because $\psi \equiv \psi_2 \wedge (\psi_1 \vee \bigcirc\psi)$, the formula ψ_2 is added to the *New* field of both nodes, ψ_1 is added to the *New* field of one node and $\bigcirc\psi$ to the *Next+* of the other node.

When the *New* field is empty, the current node is ready to be added to the set *Nodes_Set*. If there is already another node in the list with the same *Old*, *Next+*, and *Next-* fields, then the only *Incoming* field of the copy that already exists needs to be updated by adding the edges in the *Incoming* field of the current node.

If there is no such node, then the current node is added to the list *Nodes_Set*, but differently from the original algorithm [9], there are several ways how a current node is formed for its successors: if the information about the labels of the outgoing transitions is inconsistent (i.e. *Next+* is empty or there is a record r in *Next-* that is also used in a next state formula $\langle r\rangle\varphi$ in *Next+*) then there is no successor node.

Otherwise, if the formulas in the *Next+* field of the current node are only of type $\bigcirc\varphi$, then a successor node is created with a transition from the current node to the new node labeled by r for each record r not in the *Next-* field of the current node. The formulas to be satisfied by this new node are all formulas in the *Next+* field of the current node stripped off of their next state modality.

Finally, in the remaining case that there is a formula $\langle r\rangle\phi$ in *Next+* with no r in the *Next-* field, then a successor node is created with a transition labeled by r from the current node to the new node. As in the previous case, the formulas to be satisfied by this new node are all formulas in the *Next+* field of the current node stripped off by their next state modality.

The above sketched algorithm defines for every ρLTL+ formulas a BAR $BAR(\phi)$ over port names $\mathcal{N}$ and data set $\mathcal{D}$ as follows. The states are the set of nodes in *Nodes_Set*, as returned by the algorithm. Every node with the *Init* in its *Incoming* field is an initial state. The transitions of the form $q \xrightarrow{r} p$ are exactly those such that $r \in X$ for (q, X) in the *Incoming* field of p. The valuation function $V(q)(S)$ is true if and only if $Pos(q) \subseteq S$ and $S \cap Neg(q) = \emptyset$, where $Pos(q)$ and $Neg(q)$ are the sets of positive and negative occurrences of a subset of names N in the field *Old* of the node q. Finally, for each subformula $\psi_1\, U\, \psi_2$ of ψ we define an accepting state F containing all nodes q such that $\psi_1\ U\ \psi_2 \notin t(q)$ or $\psi_2 \in t(q)$, where $t(q)$ is the union of the fields *Old*, *Next+* and the set containing $\neg\langle r\rangle true$ for each record r in the field *Next-* of the node q.

Theorem 2. *Let ϕ be a ρLTL+ formula over names $\mathcal{N}$ and data $\mathcal{D}$ and $BAR(\phi)$ be the BAR produced by the above algorithm. Then, the accepted language of $BAR(\phi)$ is the set of all models of ϕ, that is $L(BAR(\phi)) = \parallel \phi \parallel$.*

As explained in the previous section, a formula about a Reo connector can be verified by (1) constructing the automaton for the negation of the formula, (2) constructing the product automaton using the model of the Reo connector, and (3) checking the resulting automaton for emptiness.

7 Conclusion

In this paper we have presented a model and a logic for the full Reo language of connectors. The model is based on ordinary Büchi automata. The logic is an extension of the linear time temporal logic with records labeling the next time operator. Records are used for modeling synchronization and mutual exclusion of port data flow. Sets of port names are used as propositions for modeling context dependencies of connectors.

We have presented extensions of the ordinary global and on the fly construction of automata for linear time temporal formulas, the basis of an automata-based approach for model checking Reo connectors. We believe that our approach opens the way to verifying Reo connectors by model checking using professional software tools like Spin [10]. Further work should investigate the possibilities that arise from exporting Reo connectors modeled as Büchi Automata into Promela code. Ideally, once a connector has been defined it should be possible for a user to verify it by exporting it in the Promela syntax, for example using a graphical frontend like GOAL [20].

References

1. Arbab, F.: Reo: a Channel-based Coordination Model for Component Composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
2. Arbab, F., Rutten, J.J.M.M.: A Coinductive Calculus of Component Connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2003. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003)
3. Arbab, F., Baier, C., de Boer, F., Rutten, J.J.M.M.: Models and Temporal Logics for Timed Component Connectors. In: Proc. of SEFM 2004, pp. 198–207. IEEE Computer Society, Los Alamitos (2004)
4. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modelling Component Connectors in Reo by Constraint Automata. Science of Computer Programming 61, 75–113 (2006)
5. Clarke, E., Grumberg, O., Peled, D.: Model Checking. The MIT Press, Cambridge (1999)
6. Clarke, D.: Reasoning about Connector Reconfiguration II: Basic reconfiguration Logic. In: Proc. of FSEN 2005. ENTCS, vol. 159, pp. 61–77. Elsevier, Amsterdam (2006)
7. Clarke, D., Costa, D., Arbab, F.: Connector Colouring I: Synchronisation and Context Dependency. Science of Computer Programming 66(3), 205–225 (2007)
8. Costa, D.: Intensional Constraint Automata. Unpublished notes (2008)
9. Gerth, R., Peled, D., Vardi, M., Wolper, P.: Simple On-the-fly Automatic Verification of Linear Temporal Logic. In: Proc. of the Int. Sym. on Protocol Specification, Testing, and Verification (PSTV 1995), pp. 3–18. Chapman & Hall, Boca Raton (1995)
10. Holzmann, G.J.: The Model Checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
11. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley, Reading (2006)
12. Izadi, M., Bonsangue, M.M.: Recasting Constraint Automata into Büchi Automata. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 156–170. Springer, Heidelberg (2008)
13. Izadi, M., Bonsangue, M.M., Clarke, D.: Modeling Component Connectors: Synchronisation and Context-Dependency. In: Proc. of SEFM 2008, pp. 303–312. IEEE Computer Society, Los Alamitos (2008)
14. Kaplan, D.M.: Regular Expressions and the Equivalence of Programs. Journal of Compiting System Science 3, 361–386 (1969)
15. Klüppelholz, S., Baier, C.: Symbolic Model Checking for Channel-based Component Connectors. In: Proc. of FOCLASA 2006. ENTCS, vol. 175(2), pp. 19–37. Elsevier, Amsterdam (2007)
16. Kozen, D.: Automata on guarded strings and applications. Matématica Contemporânea 24, 117–139 (2003)

17. Kupferman, O., Vardi, M.: Verification of Fair Transition Systems. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 372–382. Springer, Heidelberg (1996)
18. Madhusudan, P.: On the Fly Model Checking for Linear Time Temporal Logic. M.Sc. Thesis, Anna University, Madras, India (1996)
19. Remy, D.: Efficient Representation of Extensible Records. In: Proc. ACM SIGPLAN Workshop on ML and its applications, pp. 12–16 (1994)
20. Tsay, Y., Chen, Y., Tsai, M., Wu, K., Chan, W.: GOAL: A Graphical Tool for Manipulating Büchi Automata and Temporal Formulae. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 466–471. Springer, Heidelberg (2007)
21. Vardi, M.: An Automata-Theoretic Approach to Linear Temporal Logic. In: Moller, F., Birtwistle, G. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 238–266. Springer, Heidelberg (1996)
22. Vardi, M., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: Proc. of (LICS 1986), pp. 322–331 (1986)
23. Thomas, W.: Automata on Infinite Objects. In: Handbook of Theoretical Computer Science, vol. B, pp. 133–191. Elsevier, Amsterdam (1990)

A The Algorithm

In this appendix we present the pseudo code of algorithm sketched in Section 6. The algorithm constructs a graph of nodes and is called *Create_Graph*. It uses the function *Expand* which processes every node and updates the list of nodes *Nodes_Set*.

```
$Creat\_Graph(\phi)$
  $return(Expand([Name := New\_Name(), Incoming := \{Init\},$
          $New := \{\phi\}, Old := \emptyset, Next^+ := \emptyset, Next^- := \emptyset], \emptyset));$

$Expand(Node, Nodes\_Set)$
  if $Node.New = \emptyset$
     then if $\exists N \in Nodes\_Set$with $N.Old = Node.Old)$ and
          $Next^+(N) = Next^+(Node)$ and $Next^-(N) = Next^-(Node)$
          then { $Incoming(N) := Incoming(N) \cup Incoming(Node);$
               return$(Nodes\_Set);$ }
          else if $(\exists \langle r\rangle\phi, \langle r'\rangle\psi \in Next^+(Node)$ with $r \neq r')$ or
               $(\exists r \in Next^-(Node)\langle r\rangle\phi \in Next^+(Node))$
               then return$(Nodes\_Set \cup \{Node\})$
               else if $\nexists\langle r\rangle\phi \in Next^+(Node)$
                    then return$(Expand([Name := New\_Name(),$
                       $Incoming := \{(Name(Node),$
                          $Rec_{\mathcal{N}}(\mathcal{D})\backslash Next^-(Node))\}$
                       $New := StriptNexts(Next^+(Node))$
                       $old := \emptyset$
                       $Next^+ := \emptyset, Next^- := \emptyset],$
                       $Nodes\_Set \cup \{Node\}))$
                    else return$(Expand([Name := New\_Name(),$
                    $Incoming := \{(Name(Node), \{r\})\}$,
                    $New := StriptNexts(Next^{+}(Node)), Old := \emptyset$
                    $Next^{+} := \emptyset, Next^{-} := \emptyset], Nodes\_Set \cup \{Node\}))$
else let $\eta \in New(Node)$
  then $New(Node) := New(Node) \backslash \{\eta\}$;
      switch
        case $\eta = e$ or $\eta = \neg e$ or $\eta = true$ or $\eta = false$ :
          if  $\eta = false$ or $Neg(\eta) \in Old(Node)$
            then return$(Nodes\_Set)$
            else $\{Old(Node) := Old(Node) \cup \{\eta\}$;
                return$(Expand(Node, Nodes\_Set))\}$

        case $\eta = \phi U \psi$ or $\eta = \phi V \psi$ or $\eta = \phi \vee \psi$ :
          $Node_1 := [Name := New\_Name(), Incoming := Incoming(Node)$,
            $New := New(Node) \cup (New_1(\eta) \backslash Old(Node))$,
            $Old := Old(Node) \cup \{\eta\}$,
            $Next^{+} := Next^{+}(Node) \cup Next_1(\eta), Next^{-} := Next^{-}(Node)]$
          $Node_2 := [Name := New\_Name(), Incoming := Incoming(Node)$,
            $New := New(Node) \cup (New_2(\eta) \backslash Old(Node))$,
            $Old := Old(Node) \cup \{\eta\}$,
            $Next^{+} := Next^{+}(Node), Next^{-} := Next^{-}(Node)]$
          return$(Expand(Node_2, Expand(Node_1, Nodes\_Set)))$

        case $\eta = \phi \wedge \psi$ :
            $Old(Node) := Old(Node) \cup \{\eta\}$,
            $New(Node) := New(Node) \cup (\{\phi, \psi\} \backslash Old(Node))$
          return$(Expand(Node, Nodes\_Set))$

        case $\eta = \bigcirc \phi$ or $\eta = \langle r \rangle \phi$ :
            $Old(Node) := Old(Node) \cup \{\eta\}$,
            $Next^{+}(Node) := Next^{+}(Node) \cup (\eta\}$
          return$(Expand(Node, Nodes\_Set))$

        case $\eta = [r]\phi$ :
          $Node_1 := [Name := New\_Name(), Incoming := Incoming(Node)$,
            $New := New(Node), Old := Old(Node) \cup \{\eta\}$,
            $Next^{+} := Next^{+}(Node), Next^{-} := Next^{-}(Node) \cup \{r\}]$
          $Node_2 := [Name := New\_Name(), Incoming := Incoming(Node)$,
            $New := New(Node), Old := Old(Node) \cup \{\eta\}$,
            $Next^{+} := Next^{+}(Node) \cup \{\langle r \rangle \phi\}, Next^{-} := Next^{-}(Node)]$
          return$(Expand(Node_2, Expand(Node_1, Nodes\_Set)))$.
```

In the above algorithm we define $StripNexts(S) = \{\phi \mid \bigcirc \phi \in S \text{ or } \langle r \rangle \phi \in S\}$ for each set of ρLTL+ formulas S.

On the Expressiveness of Refinement Settings*

Harald Fecher[1], David de Frutos-Escrig[2], Gerald Lüttgen[3], and Heiko Schmidt[4]

[1] Albert-Ludwigs-Universität Freiburg, Germany
fecher@informatik.uni-freiburg.de
[2] Universidad Complutense Madrid, Spain
defrutos@sip.ucm.es
[3] University of York, U.K.
gerald.luettgen@cs.york.ac.uk
[4] Christian-Albrechts-Universität Kiel, Germany
hsc@informatik.uni-kiel.de

Abstract. Embedded-systems designers often use transition system-based notations for specifying, with respect to some refinement preorder, sets of deterministic implementations. This paper compares popular such refinement settings — ranging from transition systems equipped with failure-pair inclusion to disjunctive modal transition systems — regarding the sets of implementations they are able to express. The paper's main result is an expressiveness hierarchy, as well as language-preserving transformations between various settings. In addition to system designers, the main beneficiaries of this work are tool builders who wish to reuse refinement checkers or model checkers across different settings.

1 Introduction

Many of today's embedded systems employ control software that runs on specialized computer chips, performing dedicated tasks often without the need of an operating system. System designers typically specify such software using notations based on labeled transition systems: a possibly nondeterministic specification allows for a set of deterministic implementations, amenable to quality checks via testing or model checking. Verifiers benefit from the reduced state space in possibly nondeterministic abstractions from deterministic implementations. Choosing a suitable *refinement setting* for a given application in hand depends on various aspects, e.g., expressiveness, conciseness, and verification support.

In the concurrency-theory literature many refinement settings have been studied, with a focus on compositionality and full abstraction of, and logical characterizations and decision procedures for the underlying refinement preorders, see, e.g., [17] and the numerous references therein. Less attention has been paid to questions of expressiveness. In the context of top-down development, where sets of allowed implementations are specified at different design levels, it is of special interest to characterize the expressible sets of implementations. In general,

* Research support provided by DFG (FE 942/2-1, RO 1122/12-2), EPSRC (EP/E034853/1) and MEC (TIN2006-15660-C02-01, TIN2006-15578-C02-01).

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 276–291, 2010.

the more sets a formalism can describe, the more expressive it is and the more flexibility a system designer has by describing finer sets of implementations.

We perform the expressiveness comparison using language-preserving transformations, where the *language* of a refinement setting is the expressible set of deterministic implementations. This is analogous for trace-based languages, where language-preserving transformations have been developed between automata that differ in their fairness notion (Büchi, Muller, Rabin, Streett, parity), see [19] and the references therein.

Language-preserving transformations are valuable in the context of model checking, too, where abstract models reduce the size of the state space, while at the same time staying amenable to quality checks: if a property is model checked for an abstract model, then it is guaranteed to hold for each of its implementations. Therefore, a model checking tool over a refinement setting $\mathfrak{A}_1$ can be reused for another setting $\mathfrak{A}_2$ if every model from $\mathfrak{A}_2$ can be converted into an equivalent model from $\mathfrak{A}_1$ that defines the same language.

This paper studies and compares the expressiveness of almost a dozen refinement settings designed for deterministic transition systems. To do so, several intricate language-preserving transformations are developed. We also show how algorithms for checking a specification's consistency and for checking refinement can be derived from our transformations. While the expressiveness hierarchy is valuable for informing system designers on their choice of refinement setting, our transformations allow tool builders to reuse their refinement checkers or model checking algorithms across different settings.

2 Basic Notions: Refinement Settings and Expressiveness

To begin with, let $\mathcal{L}$ denote a *finite* set of possible actions (i.e., transition labels[1]), $|M|$ the cardinality of a set M, and $\mathcal{P}(M)$ its power set. M^* stands for the set of finite sequences over M, and $\cdot$ for sequence concatenation. For $R \subseteq M_1 \times M_2$, we write $m_1 R m_2$ if $(m_1, m_2) \in R$ and let $R^{-1} = \{(m_2, m_1) \mid (m_1, m_2) \in R\}$. If $X \subseteq M_1$, $Y \subseteq M_2$, we let $X \circ R = \{m_2 \in M_2 \mid \exists m_1 \in X : (m_1, m_2) \in R\}$ and $R \circ Y = \{m_1 \in M_1 \mid \exists m_2 \in Y : (m_1, m_2) \in R\}$, which are, e.g., used to describe the successors, resp. predecessors, of a transition relation. For $\rightsquigarrow \subseteq M_1 \times \mathcal{L} \times M_2$, we define the set of outgoing labels of $m_1 \in M_1$ by $\mathbf{O}_{\rightsquigarrow}(m_1) = \{a \in \mathcal{L} \mid \exists m_2 \in M_2 : m_1 \overset{a}{\rightsquigarrow} m_2\}$, and let $\overset{a}{\rightsquigarrow}$ stand for the relation $\{(m_1, m_2) \mid m_1 \overset{a}{\rightsquigarrow} m_2\}$. Relation $\rightsquigarrow$ is deterministic if $\forall m, a : |\{m\} \circ \overset{a}{\rightsquigarrow}| \leq 1$. Depending on the context, a function $f : M_1 \rightarrow M_2$ is also interpreted as a higher order function from $\mathcal{P}(M_1)$ to $\mathcal{P}(M_2)$ with $f(X) = \{f(m_1) \mid m_1 \in X\}$.

Definition 1. *A* transition system *(TS) $\mathcal{T}$ is a tuple $(S, S^0, \rightarrow)$ such that S is its set of states, $S^0 \subseteq S$ its non-empty set of initial states, and $\rightarrow \subseteq S \times \mathcal{L} \times S$ its transition relation. $\mathcal{T}$ is* finite *if $|S| < \infty$, and it is* deterministic *if $|S^0| = 1$ and $\rightarrow$ is deterministic. $\mathbf{T}_{\mathrm{det}}$ denotes the set of all deterministic transition systems. (DetTSs), which we also call* implementations.

[1] State predicates can be encoded via transition labels and are therefore omitted.

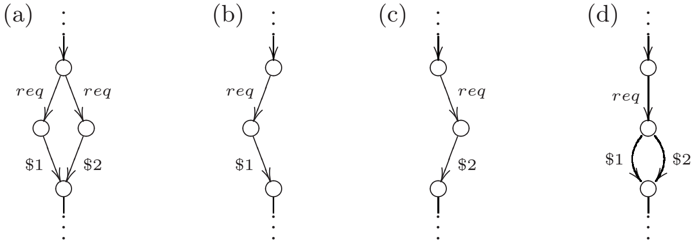


Fig. 1. Vending machine example

Note that DetTSs (up to equivalence) are the natural model for implementations in the context of open systems, where communication with the environment takes place via actions: the executions behave deterministically up to the behavior of the environment, which can only control the kind of communication (i.e., which action is executed). Since it is unusual in system modeling to specify exactly one implementation, abstract models are used to describe sets of implementations. In the context of closed systems, this is commonly done by a Kripke structure or by an automaton (it describes a set of traces, known as its language). Analogously we are looking for abstract models in the context of open systems, i.e., looking for models that describe sets of DetTSs, as illustrated by the following example:

Example 1. Consider a part of a vending machine specification, as shown in Fig. 1(a). The TS is nondeterministic, since following a *req* action it can either ask for $1 or for $2. This nondeterminism is desired, because the specification should be refinable to either a cheap machine (requesting $1, shown in Fig. 1(b)) or an expensive machine (requesting $2, shown in Fig. 1(c)). However, it depends on the employed refinement setting whether these two implementations can be modeled without also modeling the undesired implementation shown in Fig. 1(d), which gives the user the choice whether to pay $1 or $2. For instance, in failure pair semantics [4] model (a) has the undesired implementation (d), whereas in ready pair semantics [27] it has not.

Many equivalences on TSs are introduced in the literature, see [17] for an overview. Their preorders lead to different refinement notions; however, if restricted to DetTSs, they collapse as was first observed by Park [28] and further examined by Engelfriet [10]. Therefore, it is sufficient to present as equivalence notion on DetTSs only one of them, e.g., *bisimulation.*

Definition 2. *$R \subseteq S_1 \times S_2$ is a* simulation *between two TSs $\mathcal{T}_1$ and $\mathcal{T}_2$ if $\forall s_1 \in S_1^0 : \exists s_2 \in S_2^0 : (s_1, s_2) \in R$, and for all $(s_1, s_2) \in R$, $a \in \mathcal{L}$ we have $\forall s_1' \in \{s_1\} \circ \xrightarrow{a}_1 : \exists s_2' \in \{s_2\} \circ \xrightarrow{a}_2 : s_1' R s_2'$, which can equivalently be written as $(\{s_1\} \circ \xrightarrow{a}_1) \subseteq R \circ (\{s_2\} \circ \xrightarrow{a}_2)$. We say that $\mathcal{T}_1$* is simulated by *$\mathcal{T}_2$ if there is a simulation R between $\mathcal{T}_1$ and $\mathcal{T}_2$. Further, we say that $\mathcal{T}_1$ and $\mathcal{T}_2$ are* bisimilar *(or simply* equivalent*), and then write $\mathcal{T}_1 \equiv \mathcal{T}_2$, if there is a simulation R between $\mathcal{T}_1$ and $\mathcal{T}_2$ such that R^{-1} is a simulation between $\mathcal{T}_2$ and $\mathcal{T}_1$.*

We can now define refinement settings as families of *models*, into which DetTSs are embedded, with an order relating them.

Definition 3. *A* refinement setting $\mathfrak{A}$ *is a tuple* $(A, A^f, \preceq, h)$*, where* A *is a set of so called* models*,* $A^f \subseteq A$ *is a distinguished subclass of so called* finite models*,* $\preceq$ *is a preorder on* A*, called* refinement*, and* $h : \mathbf{T}_{\text{det}} \to A$ *is an* embedding*, i.e.,* $\forall \mathcal{T}_1, \mathcal{T}_2 \in \mathbf{T}_{\text{det}} : \mathcal{T}_1 \equiv \mathcal{T}_2 \Leftrightarrow h(\mathcal{T}_1) \preceq h(\mathcal{T}_2)$*. The* language $\mathfrak{A}(\alpha)$ *of a model* $\alpha \in A$ *(also called its* set of implementations *or its* possible worlds*) is the set of refining implementations of* α*, i.e.,* $\{\mathcal{T} \in \mathbf{T}_{\text{det}} \mid h(\mathcal{T}) \preceq \alpha\}$*.*

Though this is not required by the definition, it is best to first think of $h(\mathbf{T}_{\text{det}})$ as the bottom elements of the refinement preorder $\preceq$. They correspond to the implementations. Then, equivalence on DetTSs ($\mathcal{T}_1 \equiv \mathcal{T}_2$) must imply "refinement equivalence", i.e., $h(\mathcal{T}_1) \preceq h(\mathcal{T}_2)$, and, directly implied by equivalence of $\equiv$, $h(\mathcal{T}_2) \preceq h(\mathcal{T}_1)$ on these bottom elements. For the other direction, refinement between models on the implementation level must be enough to establish equivalence on DetTSs, which makes sure that every implementation can be specified alone, without any other, non-equivalent refining implementations.

Now in fact, the definition also allows non-implementations below implementation level, i.e., below elements of $h(\mathbf{T}_{\text{det}})$. These appear, e.g., in (disjunctive) mixed transition systems [26] and are unsatisfiable, i.e., have an empty language. Our notion of expressiveness is based on the expressible languages of a refinement setting:

Definition 4. *Let* $\mathfrak{A} = (A, A^f, \preceq, h)$ *be a refinement setting. A* language-preserving transformation *from* $\mathfrak{A}_1$ *to* $\mathfrak{A}_2$ *is a total function* $f : A_1^f \to A_2^f$ *such that* $\mathfrak{A}_1(\alpha) = \mathfrak{A}_2(f(\alpha))$ *for all* $\alpha \in A_1^f$*. We say that* $\mathfrak{A}_1$ *is* at least as expressive as $\mathfrak{A}_2$ *if there is a language-preserving transformation from* $\mathfrak{A}_2$ *to* $\mathfrak{A}_1$*.*

Reconsider Ex. 1 where we claimed that model (a) expresses, with respect to ready pair semantics, implementations (b) and (c), whereas it also has the further implementation (d) with respect to failure pair semantics. If we show that there is also no other specification that expresses exactly (b) and (c) (up to equivalence) in failure pair semantics, we know that there can be no language-preserving transformation from ready pair semantics to failure pair semantics. To prove that ready pair semantics is more expressive than failure pair semantics, we also have to prove that every language expressible in failure pair semantics can also be expressed in ready pair semantics.

By Def. 4, language-preserving transformations are mapping *finite* abstract models to *finite* abstract models (though implementations may be infinite). We are especially interested in such mappings because they preserve the (direct) amenability to applications like model checking. Furthermore, expressiveness results for *infinite* models are mostly trivial, because infinite initial state sets can be used to describe any desired language (by dedicating one initial state to each desired implementation).

3 A Wide Collection of Refinement Settings

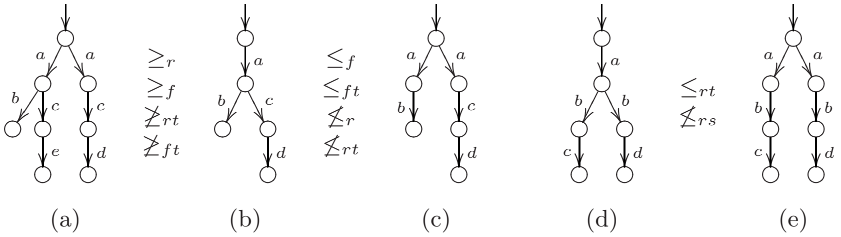
This section recalls popular refinement settings that have been studied in the literature, where models are either TSs, *synchronously communicating* TSs [12], *modal/mixed* TSs [25,7], or *disjunctive modal/mixed* TSs [26]. It is easily checked that all these settings are indeed refinement settings.

Transition systems. First note that TS-based models equipped with trace inclusion or simulation, when taking h as the identity function, do not yield refinement settings, since DetTSs cannot be embedded into these refinement preorders, i.e., DetTSs $\mathcal{T}_1$, $\mathcal{T}_2$ can be found such that $h(\mathcal{T}_1) \preceq h(\mathcal{T}_2)$ but $\mathcal{T}_1 \not\equiv \mathcal{T}_2$, e.g., $\mathcal{T}_1 = \rightarrow\!\circ$ and $\mathcal{T}_2 = \rightarrow\!\circ\!\xrightarrow{a}\!\circ$. Therefore, preorders in refinement settings must preserve, in both directions, the enabledness of actions when comparing DetTSs, e.g., every refinement of $\mathcal{T}_2$ above must have action a enabled in its root state, which $\mathcal{T}_1$ has not. We present refinement settings based on *failure pairs* (also called failures) [4], *failure traces* (also called refusal) [29], *ready pairs* (also called readiness) [27], *ready traces* [1], *possible worlds* [32], and *ready simulations* [2][2]. For a TS $\mathcal{T}$,

- a ready trace of $s \in S$ is a trace starting in s, together with the sets of enabled actions after every subtrace. Formally, the set of *ready traces* of $\mathcal{T}$ is the smallest set $\mathrm{Tr}^{\mathcal{T}}_{\mathrm{RT}} \subseteq S \times ((\mathcal{P}(\mathcal{L}) \cdot \mathcal{L})^* \cdot \mathcal{P}(\mathcal{L}))$ with $(s, \mathbf{O}_{\rightarrow}(s)) \in \mathrm{Tr}^{\mathcal{T}}_{\mathrm{RT}}$ and $((s', \sigma) \in \mathrm{Tr}^{\mathcal{T}}_{\mathrm{RT}} \wedge s \xrightarrow{a} s') \Rightarrow (s, \mathbf{O}_{\rightarrow}(s) a \sigma) \in \mathrm{Tr}^{\mathcal{T}}_{\mathrm{RT}}$, for any $s, s' \in S$, $a \in \mathcal{L}$.
- a ready pair of $s \in S$ is a trace starting in s, together with the set of enabled actions after the complete trace. Formally, the set of *ready pairs* $\mathrm{Tr}^{\mathcal{T}}_{\mathrm{R}} \subseteq S \times (\mathcal{L}^* \cdot \mathcal{P}(\mathcal{L}))$ is the set of traces from $\mathrm{Tr}^{\mathcal{T}}_{\mathrm{RT}}$ where each but the last element from $\mathcal{P}(\mathcal{L})$ is removed.
- a failure pair of $s \in S$ is a trace starting in s, together with a set of actions that are not enabled after the complete trace. Formally, the set of *failure pairs* $\mathrm{Tr}^{\mathcal{T}}_{\mathrm{F}} \subseteq S \times (\mathcal{L}^* \cdot \mathcal{P}(\mathcal{L}))$ is the set of traces that can be obtained by replacing the last element (the one from $\mathcal{P}(\mathcal{L})$) in a trace from $\mathrm{Tr}^{\mathcal{T}}_{\mathrm{R}}$ by any subset of its complement.
- a failure trace of $s \in S$ is a trace starting in s, together with, for every subtrace, a set of actions that are not enabled after the subtrace. Formally, the set of *failure traces* $\mathrm{Tr}^{\mathcal{T}}_{\mathrm{FT}} \subseteq S \times ((\mathcal{P}(\mathcal{L}) \cdot \mathcal{L})^* \cdot \mathcal{P}(\mathcal{L}))$ is the set of traces that can be obtained by replacing every element in $\mathcal{P}(\mathcal{L})$ from a trace in $\mathrm{Tr}^{\mathcal{T}}_{\mathrm{RT}}$ by any subset of its complement.

Now the refinement settings of *ready pair inclusion* $\mathbb{T}_{\mathrm{r}}$, *ready trace inclusion* $\mathbb{T}_{\mathrm{rt}}$, *failure pair inclusion* $\mathbb{T}_{\mathrm{f}}$, *failure trace inclusion* $\mathbb{T}_{\mathrm{ft}}$, *ready simulation* $\mathbb{T}_{\mathrm{rs}}$ and *possible worlds inclusion* $\mathbb{T}_{\mathrm{pw}}$ consist of TSs, the identity embedding on DetTSs, and the refinement notion given by ready pair inclusion: $(S^0_1 \circ \mathrm{Tr}^{\mathcal{T}_1}_{\mathrm{R}}) \subseteq (S^0_2 \circ \mathrm{Tr}^{\mathcal{T}_2}_{\mathrm{R}})$; resp. ready trace inclusion: $(S^0_1 \circ \mathrm{Tr}^{\mathcal{T}_1}_{\mathrm{RT}}) \subseteq (S^0_2 \circ \mathrm{Tr}^{\mathcal{T}_2}_{\mathrm{RT}})$; resp. failure pair inclusion: $(S^0_1 \circ \mathrm{Tr}^{\mathcal{T}_1}_{\mathrm{F}}) \subseteq (S^0_2 \circ \mathrm{Tr}^{\mathcal{T}_2}_{\mathrm{F}})$; resp. failure trace inclusion: $(S^0_1 \circ \mathrm{Tr}^{\mathcal{T}_1}_{\mathrm{FT}}) \subseteq (S^0_2 \circ \mathrm{Tr}^{\mathcal{T}_2}_{\mathrm{FT}})$; resp. ready simulation: $\mathcal{T}_1$ is *ready simulated by* $\mathcal{T}_2$ if there exists a simulation R

[2] For space reasons, compact definitions, conforming with the standard ones, are used.



(a) (b) (c) (d) (e)

Fig. 2. Illustration of the refinement preorders on TSs, where $\leq_x$ stands for refinement with respect to refinement notion x. These examples are derived from Counterexamples 5, 6, and 8 of [17].

between $\mathcal{T}_1$ and $\mathcal{T}_2$ (i.e., $\mathcal{T}_1$ is simulated by $\mathcal{T}_2$) such that the enabled actions are the same for related elements, i.e., $(s_1, s_2) \in R \Rightarrow \mathbf{O}_{\rightarrow_1}(s_1) = \mathbf{O}_{\rightarrow_2}(s_2)$; resp. possible worlds inclusion: $\mathbb{T}_{\mathrm{rs}}(\mathcal{T}_1) \subseteq \mathbb{T}_{\mathrm{rs}}(\mathcal{T}_2)$. Fig. 2 illustrates some differences between the refinement notions. In all these cases, and also in the forthcoming ones when we will consider other more sophisticated classes of transition systems, a system is finite if and only if its set of states is finite.

Synchronously-communicating transition systems. Synchronously-communicating TSs [12] extend TSs by a predicate $e(s)$ on states s that indicates which actions must be present (i.e., enabled) at s and thus cannot be removed by refinements. Formally, a *synchronously-communicating transition system* (STS) without fairness is a tuple $(\mathcal{T}, e)$ such that $\mathcal{T}$ is a TS and $e : S \rightarrow \mathcal{P}(\mathcal{L})$ is its *existence predicate*. It is *must-saturated* if $a \in e(s)$ implies the existence of an outgoing transition labeled by a, i.e., $\forall s \in S : e(s) \subseteq \mathbf{O}_{\rightarrow}(s)$. For the definition of the refinement settings $\mathbb{S}$ of *STSs* and $\mathbb{S}_{\mathrm{ms}}$ of *must-saturated STSs*, DetTSs are embedded by taking e to be $\mathbf{O}_{\rightarrow}$, and $(\mathcal{T}_1, e_1)$ refines $(\mathcal{T}_2, e_2)$ if there exists a simulation R between $\mathcal{T}_1$ and $\mathcal{T}_2$ such that $(s_1, s_2) \in R \Rightarrow e_2(s_2) \subseteq e_1(s_1)$. For example, $\rightarrow\!\bigcirc\xrightarrow{a}\bigcirc\xrightarrow{a}\bigcirc$ is an implementation of the STS on the left of Fig. 3, whereas $\rightarrow\!\bigcirc\xrightarrow{a}\bigcirc\xrightarrow{b}\bigcirc$ is not.

Modal/mixed transition systems. Mixed TSs [7] have *must-transitions* (that must be present in an implementation) and *may-transitions* (nothing else may be present in an implementation). A modal TS [25] has the additional requirement that every must-transition also has to be a may-transition. Formally, a *mixed transition system* is a tuple $(\mathcal{T}, \hookrightarrow)$ such that $\mathcal{T}$ is a TS, where its transition relation is called *may*-transition relation here, and $\hookrightarrow \ \subseteq S \times \mathcal{L} \times S$ is its *must*-transition relation. It is a *modal transition system* if $\hookrightarrow \ \subseteq \ \rightarrow$. For the definition of the refinement settings $\mathbb{M}$ of *mixed TSs* and $\mathbb{M}_{\mathrm{mod}}$ of *modal TSs*, DetTSs are embedded by taking $\rightarrow$ as the must-transition relation, and $(\mathcal{T}_1, \hookrightarrow_1)$ refines $(\mathcal{T}_2, \hookrightarrow_2)$ if there exists a simulation R between $\mathcal{T}_1$ and $\mathcal{T}_2$ such that R^{-1} is a simulation between $(S_2, \emptyset, \hookrightarrow_2)$ and $(S_1, \emptyset, \hookrightarrow_1)$.

Modal/mixed TSs, as well as their disjunctive variants presented in the following, were originally designed for general transition systems as implementations.

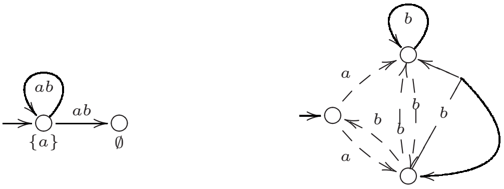


Fig. 3. An example STS (left) and disjunctive mixed TS (right)

Interpreting them with respect to DetTS leads to new kinds of modeling techniques, which improve the succinctness of these settings, allowing for more compact representations, as, e.g., discussed in [14]. For example, two a-labeled must-transitions from the same state leading to states s_1 and s_2 require any implementation to implement, after the only possible a-step, both the behavior of s_1 *and* s_2. We call such behavior *conjunctive behavior.*

Disjunctive modal/mixed transition systems. Disjunctive modal/mixed TSs [26] generalize modal/mixed TSs by introducing hypertransitions that point to sets of states rather than single states. A *must-hypertransition* t indicates that the implementation must have a transition with corresponding label to a state that is related to at least one element in the target set of t, i.e., the targets are interpreted disjunctively. We present disjunctive modal TSs as a special case of their mixed version, where must-hypertransitions need not necessarily occur as may-transitions: a *disjunctive mixed transition system* is a tuple $(\mathcal{T}, \mapsto)$ such that $\mathcal{T}$ is a TS, where its transition relation is called *may*-transition relation here and $\mapsto\ \subseteq S \times \mathcal{L} \times \mathcal{P}(S)$ is its *must*-hypertransition relation. It is a *disjunctive modal transition system* if all must-hypertransition target sets are non-empty and only have elements that are also targets of may-transitions, i.e., $\forall s \in S, a \in \mathcal{L}, \ddot{S} \in \{s\}\circ \overset{a}{\mapsto}$: $\ddot{S} \neq \emptyset \wedge \ddot{S} \subseteq \{s\}\circ \xrightarrow{a}$. For the definition of the refinement settings $\mathbb{D}$ of *disjunctive mixed TSs* and $\mathbb{D}_{\mathrm{mod}}$ of *disjunctive modal TSs*, DetTSs are embedded by taking $\{(s, a, \{s'\}) \mid s \xrightarrow{a} s'\}$ as must-hypertransitions, and $(\mathcal{T}_1, \mapsto_1)$ refines $(\mathcal{T}_2, \mapsto_2)$ if there is a simulation R between $\mathcal{T}_1$ and $\mathcal{T}_2$ such that $\forall (s_1, s_2) \in R$, $a \in \mathcal{L}, \ddot{S}_2 \in \{s_2\}\circ \overset{a}{\mapsto}_2$: $\exists \ddot{S}_1 \in \{s_1\}\circ \overset{a}{\mapsto}_1$: $\forall s_1' \in \ddot{S}_1 : \exists s_2' \in \ddot{S}_2 : s_1' R s_2'$. For example, $\rightarrow\circ\xrightarrow{a}\circ\circlearrowleft b$ is an implementation of the disjunctive mixed TS on the right of Fig. 3, whereas $\rightarrow\circ\xrightarrow{a}\circ\xrightarrow{b}\circ$ is not.

4 Comparison

This section establishes an expressiveness hierarchy constructively by presenting language-preserving transformations or showing their non-existence by counterexample. In particular, we have paid attention to *simple* transformations and *small-sized* transformed models. All transformations also work for infinite-state systems but, not surprisingly, their mappings are not guaranteed to be finite.

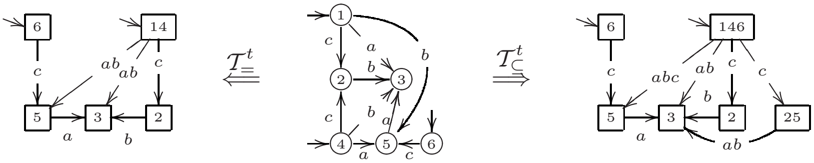
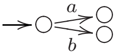
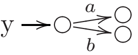


Fig. 4. Illustration of the transformations' images $\mathcal{T}^t_{=}$ and $\mathcal{T}^t_{\subseteq}$. Targets of transitions without source indicate initial states. Transitions having a set as label indicate a set of transitions, one for each label. The numbers of the state names in the left and right systems correspond to the state subset encoding.

To begin with, the identity function is a transformation from $\mathbb{S}_{\text{ms}}$ into $\mathbb{S}$; from $\mathbb{M}_{\text{mod}}$ into $\mathbb{M}$; from $\mathbb{D}_{\text{mod}}$ into $\mathbb{D}$; from $\mathbb{T}_{\text{pw}}$ into $\mathbb{T}_{\text{rs}}$; and from $\mathbb{T}_{\text{rs}}$ into $\mathbb{T}_{\text{pw}}$.

Trace inclusions. Due to the coinductive definition of simulation, checking refinement in simulation-based settings only depends on what remains to be considered in the future, e.g.,  is not an implementation of $\mathcal{T}_{\text{rs,rt}}$ of Fig. 6 in simulation-like approaches, because the refinement relation has to decide for one of the two initial states. This is different in trace-like approaches, where at any time it is possible to go back in a trace and resolve nondeterminism differently, as long as the traces still coincide. Consequently  *is* a refinement of $\mathcal{T}_{\text{rs,rt}}$ in trace-like settings. A transformation from a trace-like setting to $\mathbb{T}_{\text{rs}}$ therefore has to make every previous nondeterministic choice explicit in the state space. Hence, power sets over states are used in the transformations from $\mathbb{T}_{\text{rt}}$, resp. from $\mathbb{T}_{\text{ft}}$, as illustrated in Fig. 4.

Transformation 1. *For any TS $\mathcal{T}$, $\mathbb{T}_{\text{rt}}(\mathcal{T}) = \mathbb{T}_{\text{rs}}(\mathcal{T}^t_{=}) = \mathbb{T}_{\text{rt}}(\mathcal{T}^t_{=})$ and $\mathbb{T}_{\text{ft}}(\mathcal{T}) = \mathbb{T}_{\text{rs}}(\mathcal{T}^t_{\subseteq}) = \mathbb{T}_{\text{rt}}(\mathcal{T}^t_{\subseteq}) = \mathbb{T}_{\text{ft}}(\mathcal{T}^t_{\subseteq})$ with*

$$\mathcal{T}^t_{\lhd} = (\mathcal{P}(S), \Psi^t_{\lhd}(S^0), \{(\ddot{S}, a, \ddot{S}') \mid a \in \mathcal{L} \wedge \ddot{S}' \in \Psi^t_{\lhd}(\ddot{S} \circ \xrightarrow{a})\})\ ,\ \textit{where}$$
$$\lhd\ \in \{=, \subseteq\}\quad \textit{and}\quad \Psi^t_{\lhd}(\widehat{S}) = \{\{s \in \widehat{S} \mid \mathbf{O}_{\rightarrow}(s) \lhd L\} \mid L \subseteq \mathcal{L}\} \setminus \{\emptyset\}.$$

Transformations from pair to trace approaches are similar, except that all reachable states with respect to the underlying label trace are collected (since a failure/ready pair has as history information only the underlying label trace and no intermediate failure/ready sets). Hence, pairs of original states and allowed labels are the state set of these transformations, as illustrated in Fig. 5.

Transformation 2. *For any TS $\mathcal{T}$, $\mathbb{T}_{\text{r}}(\mathcal{T}) = \mathbb{T}_{\text{rs}}(\mathcal{T}^p_{=}) = \mathbb{T}_{\text{rt}}(\mathcal{T}^p_{=}) = \mathbb{T}_{\text{r}}(\mathcal{T}^p_{=})$ and $\mathbb{T}_{\text{f}}(\mathcal{T}) = \mathbb{T}_{\text{rs}}(\mathcal{T}^p_{\subseteq}) = \mathbb{T}_{\text{rt}}(\mathcal{T}^p_{\subseteq}) = \mathbb{T}_{\text{ft}}(\mathcal{T}^p_{\subseteq}) = \mathbb{T}_{\text{r}}(\mathcal{T}^p_{\subseteq}) = \mathbb{T}_{\text{f}}(\mathcal{T}^p_{\subseteq})$ with*

$$\mathcal{T}^p_{\lhd} = (\mathcal{P}(S) \times \mathcal{P}(\mathcal{L}), \Psi^p_{\lhd}(S^0), \{((\ddot{S}, L), a, Z') \mid a \in L \wedge Z' \in \Psi^p_{\lhd}(\ddot{S} \circ \xrightarrow{a})\}),$$
$$\textit{where}\ \lhd\ \in \{=, \subseteq\}\quad \textit{and}\quad \Psi^p_{\lhd}(\widehat{S}) = \{(\widehat{S}, \widehat{L}) \mid \exists s \in \widehat{S} : \mathbf{O}_{\rightarrow}(s) \lhd \widehat{L}\}.$$

The increase of expressiveness of these settings is illustrated in Fig. 10: the failure approach *cannot* express an exclusive alternative between two labels.

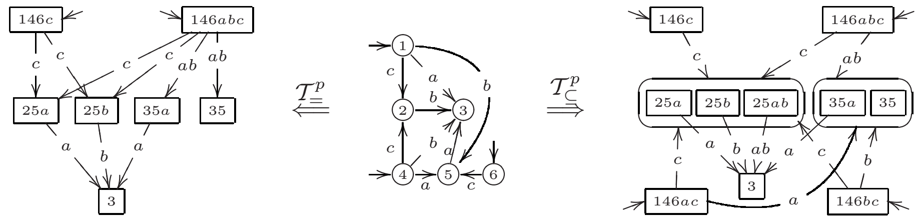


Fig. 5. Illustration of the transformations' images $\mathcal{T}^p_{=}$ and $\mathcal{T}^p_{\subseteq}$. In the right picture, states that have the same targets are identified. A transition pointing to an oval indicates a set of transitions pointing to each element inside the oval. The numbers (resp. labels) of the state names in the left and right pictures correspond to the state (resp. label) subset encoding in the respective transformation.

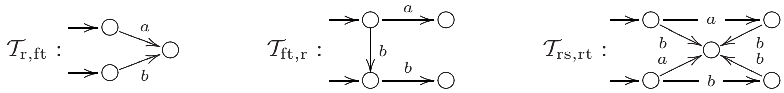
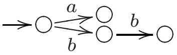
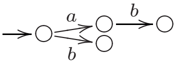
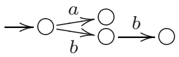
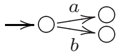

$\mathcal{T}_{\mathrm{r,ft}}$:
a
b
$\mathcal{T}_{\mathrm{ft,r}}$:
a
b
b
$\mathcal{T}_{\mathrm{rs,rt}}$:
a
b
a
b
b
b

Fig. 6. TSs illustrating increases of expressiveness

However, the ready approach can do it, as shown by $\mathcal{T}_{\mathrm{r,ft}}$ in Fig. 6. This is reflected by the axiom $c.P_a + c.P_b \equiv c.P_a + c.P_b + c(P_a + P_b)$, which is valid for failure semantics, but not for the semantics based on ready sets. In pair approaches, behavior can only be described up to alternatives having the same label path histories, whereas a trace approach can also distinguish alternatives that have the same label path history but different next-step possibilities (up to failure or ready interpretation). For example, no TS with respect to failure pair, resp. ready pair, can have  and $\rightarrow\bigcirc\xrightarrow{b}\bigcirc$ as implementations, without also having $\rightarrow\bigcirc\xrightarrow{b}\bigcirc\xrightarrow{b}\bigcirc$ as implementation. However, $\mathcal{T}_{\mathrm{ft,r}}$ of Fig. 6 defines such a language via failure trace (resp. ready trace). Ready simulation increases the expressiveness even more by distinguishing also alternatives with the same label path history and next-step possibilities, but different future behaviors in the past. For example, no TS with respect to a trace approach can have  and  as implementations, without also having . However, $\mathcal{T}_{\mathrm{rs,rt}}$ of Fig. 6 defines such a language via ready simulation. The following lemma summarizes the above results, from which the 'strictly greater' expressiveness results for the TS-based settings are derived by transitivity arguments.

Lemma 1. *For $\mathcal{T}_{\mathrm{r,ft}}$ $\mathcal{T}_{\mathrm{ft,r}}$, $\mathcal{T}_{\mathrm{rs,rt}}$ in Fig. 6 and arbitrary $\mathcal{T}$, we have:*

$$\mathbb{T}_{\mathrm{r}}(\mathcal{T}_{\mathrm{r,ft}}) \neq \mathbb{T}_{\mathrm{ft}}(\mathcal{T}),\ \mathbb{T}_{\mathrm{ft}}(\mathcal{T}_{\mathrm{ft,r}}) \neq \mathbb{T}_{\mathrm{r}}(\mathcal{T}) \textit{ and } \mathbb{T}_{\mathrm{rs}}(\mathcal{T}_{\mathrm{rs,rt}}) \neq \mathbb{T}_{\mathrm{rt}}(\mathcal{T}).$$

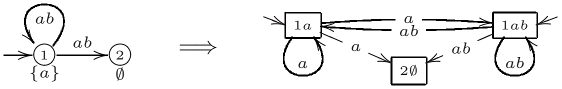


Fig. 7. Example of the transformation from $\mathbb{S}_{\mathrm{ms}}$ to $\mathbb{T}_{\mathrm{rs}}$. For STSs, the image of e is depicted close to the state. The numbers (resp. labels) of the state names in the right picture correspond to the state (resp. label) subset encoding of the transformation.

The following proposition shows how our transformations and the efficient decision procedures for simulation-like preorders [6] can be used to decide the corresponding inclusion problems. However, in general such derived algorithms would have limited practical relevance since deciding trace-like preorders is PSPACE-complete [31], but in some particular cases the complexity of the decision procedure is certainly much lower.

Proposition 1. *$\mathcal{T}$ is ready trace (failure trace, ready pair, failure pair) included in $\widetilde{\mathcal{T}}$ iff $\mathcal{T}^t_=$ (resp. $\mathcal{T}^t_\subseteq$, $\mathcal{T}^p_=$, $\mathcal{T}^p_\subseteq$) is ready simulated by $\widetilde{\mathcal{T}}^t_=$ (resp. $\widetilde{\mathcal{T}}^t_\subseteq$, $\widetilde{\mathcal{T}}^p_=$, $\widetilde{\mathcal{T}}^p_\subseteq$).*

Ready simulation and must-saturated STS. $\mathbb{T}_{\mathrm{rs}}$ is transformed to $\mathbb{S}_{\mathrm{ms}}$ by setting the existence predicate to the set of labels for which an outgoing transition exists.

Transformation 3. *For any TS $\mathcal{T}$, $\mathbb{T}_{\mathrm{rs}}(\mathcal{T}) = \mathbb{S}_{\mathrm{ms}}((\mathcal{T}, \mathbf{O}_\rightarrow))$.*

For transforming $\mathbb{S}_{\mathrm{ms}}$ to $\mathbb{T}_{\mathrm{rs}}$, every state s is combined with a ready set $L \subseteq \mathcal{L}$, indicating that exactly these labels may *not* be removed. The incoming transitions are determined by the incoming ones of s. Fig. 7 presents a simple example.

Transformation 4. *For any must-saturated STS $(\mathcal{T}, e)$, $\mathbb{S}_{\mathrm{ms}}((\mathcal{T}, e)) = \mathbb{T}_{\mathrm{rs}}((S', S' \cap (S^0 \times \mathcal{P}(\mathcal{L})), \rightarrow'))$ with*

$$S' = \{(s, L) \mid e(s) \subseteq L \subseteq \mathbf{O}_\rightarrow(s)\}, \quad \rightarrow' = \{((s, L), a, (s', L')) \mid s \xrightarrow{a} s' \wedge a \in L\}.$$

$\mathbb{S}$ is indeed strictly more expressive than $\mathbb{T}_{\mathrm{rs}}$, because the latter does not allow one to specify the empty language.

Lemma 2. *For the STS $\rightarrow\underset{\{a\}}{\bigcirc}$ and arbitrary $\mathcal{T}$, we have $\mathbb{S}(\rightarrow\underset{\{a\}}{\bigcirc}) \neq \mathbb{T}_{\mathrm{rs}}(\mathcal{T})$.*

If we allow the initial set of a TS to be empty and therefore not to have any DetTS as refinement, we obtain, by the following algorithm, that $\mathbb{S}$ and $\mathbb{S}_{\mathrm{ms}}$ are equally expressive. Hence, the empty set is the only language that increases the expressive power of $\mathbb{S}$ and any other equally expressive refinement setting.

Proposition 2. *An STS can be linearly transformed to an equivalent must-saturated one (possibly with an empty initial state set) by successively removing those states s and their in- and outgoing transitions, for which $e(s) \not\subseteq \mathbf{O}_\rightarrow(s)$.*

Remaining settings. $\mathbb{S}$ is transformed to $\mathbb{M}$ by modeling predicate e via must-transitions to a special state s_{all} that is refined by each implementation state.

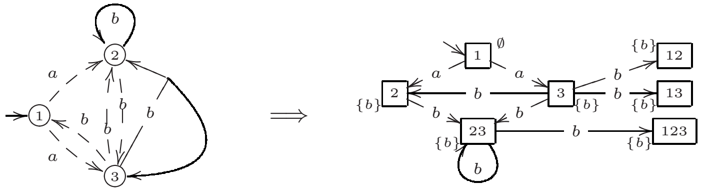


Fig. 8. Example of the transformation from $\mathbb{D}$ to $\mathbb{S}$. For disjunctive mixed TSs, solid (dashed) arrows model must-transitions (resp. may-transitions). Branching solid arrows model must-hypertransitions. The numbers of the state names in the right picture correspond to the state subset encoding; e.g., the self loop of state $\{2,3\}$ is obtained by choosing g and h such that $g(2)=3$, $g(3)=2$, $h(\{2\})=2$, and $h(\{2,3\})=2$.

Transformation 5. *For any STS* $(\mathcal{T},e)$*,* $\mathbb{S}((\mathcal{T},e)) = \mathbb{M}(((S \cup \{s_{\mathrm{all}}\}, S^0, \rightarrow \cup (\{s_{\mathrm{all}}\} \times \mathcal{L} \times \{s_{\mathrm{all}})\})), \hookrightarrow'))$ *with* $s_{\mathrm{all}} \notin S$ *and* $\hookrightarrow' = \bigcup_{s \in S} \{s\} \times e(s) \times \{s_{\mathrm{all}}\}$.

$\mathbb{M}$ (resp. $\mathbb{M}_{\mathrm{mod}}$) is transformed to $\mathbb{D}$ (resp. $\mathbb{D}_{\mathrm{mod}}$) by turning each must-transition pointing to s into a must-hypertransition pointing to $\{s\}$.

Transformation 6. *Let* $\mapsto' = \{(s, a, \{s'\}) \mid s \overset{a}{\hookrightarrow} s'\}$*. Then for any mixed TS* $(\mathcal{T}, \hookrightarrow)$*,* $\mathbb{M}((\mathcal{T}, \hookrightarrow)) = \mathbb{D}((\mathcal{T}, \mapsto'))$*, and for any modal transition system* $(\mathcal{T}, \hookrightarrow)$*,* $\mathbb{M}_{\mathrm{mod}}((\mathcal{T}, \hookrightarrow)) = \mathbb{D}_{\mathrm{mod}}((\mathcal{T}, \mapsto'))$*.*

We proceed with the transformation from $\mathbb{D}$ to $\mathbb{S}$. The new states are subsets $\ddot{S} \subseteq S$, with the intuition that a related implementation state has to be related to all elements of $\ddot{S}$. Transitions from $\ddot{S}$ lead to those subset states that consist of a combination of targets of must-hypertransitions from states $s \in \ddot{S}$, together with one may-target for each $s \in \ddot{S}$. In the definition of these successor sets $C^a_{\ddot{S}}$, we use choice functions $h : \mathcal{P}(S) \rightarrow S$ for the selection of an element from a must-hypertransition target, and $g : S \rightarrow S$ for the selection of a may-transition target. The existence predicate holds for a at $\ddot{S}$ iff there is a must-hypertransition with label a and leaving a state in $\ddot{S}$. Fig. 8 shows an example of this transformation.

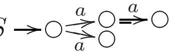
Transformation 7. *For any disjunctive mixed TS* $(\mathcal{T}, \mapsto)$*,* $\mathbb{D}((\mathcal{T}, \mapsto)) = \mathbb{S}(((\mathcal{P}(S), \{\{s^0\} \mid s^0 \in S^0\}, \bigcup_{\ddot{S} \subseteq S, a \in \mathcal{L}} \{\ddot{S}\} \times \{a\} \times C^a_{\ddot{S}}), \mathbf{O}_{\mapsto}))$ *with* $C^a_{\ddot{S}} = \{g(\ddot{S}) \cup h(\ddot{S} \circ \overset{a}{\mapsto}) \mid \forall s \in \ddot{S} : s \xrightarrow{a} g(s) \wedge \forall \ddot{S}' \in (\ddot{S} \circ \overset{a}{\mapsto}) : h(\ddot{S}') \in \ddot{S}'\}$*.*

Finally, we present the transformation from $\mathbb{D}$ to $\mathbb{M}_{\mathrm{mod}}$ with complexity $\mathcal{O}((2^{|S|})^{|\mathcal{L}|})$ and, by restriction to $\mathbb{M}$, obtain a transformation from $\mathbb{M}$ into $\mathbb{M}_{\mathrm{mod}}$ with complexity $\mathcal{O}(|S|^{|\mathcal{L}|})$.

A first observation is that $\mathbb{M}_{\mathrm{mod}}$ can represent conjunctive behavior since the properties described by all must-transition targets have to hold after the step. However, a state in $\mathbb{M}_{\mathrm{mod}}$ cannot enforce the existence of a label a and, at the same time, model disjunctive behavior after the execution of a (via non-determinism) because, as soon as an outgoing must-transition (with implicit may-transition) exists, all further outgoing may-transitions with the same label are redundant: in deterministic refinements, the unique transition of the implementation already has to match with the may-transition corresponding to the must-transition. The solution is to distribute the needed requirements to multiple states, where one state $(s, -)$ enforces action existence, and several further states $(s, \ddot{S})$, with $\ddot{S} \subseteq S$, encode the nondeterministic behavior. Must-transitions to all these states make sure that each of them is related to a single implementation state, which therefore has to meet all of the requirements. These must-transitions originate from another kind of state $(s, f) \in S \times \mathrm{F}_s$, which encodes a complete resolution of the next-step nondeterminism in the $\mathbb{D}$-system. Here, F_s is a set of functions that collect, per label a, a set from $C^a_{\{s\}}$, i.e., an element from every must-hypertransition target together with one may-transition target. The resulting collection contains those $\mathbb{D}$-states to which the successor of a related implementation state must be related. To be precise, no element can be collected if no must-transition is present $(a \notin \mathbf{O}_{\mapsto}(s) \wedge g(a) = \emptyset)$. For contradictory states, F_s is empty. A state (s, f) points, via a-labeled must-transitions, to every element of $g(a) \times (\{-\} \cup \mathcal{P}(S))$. A state $(s, -)$ encodes the labels necessary in s via must-transitions to state s_{all}, which is refined by any implementation state. A state $(s, \ddot{S})$ is used to model the nondeterministic behavior of (i) the must-hypertransition target $\ddot{S}$, or (ii) the may-transitions if $\ddot{S} = \{s\} \circ \xrightarrow{a}$. This is achieved by outgoing may-transitions to every element s' of $\ddot{S}$, combined with any value of $\mathrm{F}_{s'}$. For technical reasons, $(s, \ddot{S})$ points to s_{all} if $\ddot{S}$ does not correspond to a must-hypertransition target or to the may-transition targets. Fig. 9 shows an example of this transformation.

Transformation 8. *For any disjunctive mixed TS* $(\mathcal{T}, \mapsto)$, $\mathbb{D}((\mathcal{T}, \mapsto)) = \mathbb{M}_{\text{mod}}(((S', {S^0}', \rightarrow'), \hookrightarrow'))$ *with* $C^a_{\{s\}}$ *as in Transf. 7 and*

$$\begin{aligned}
\mathrm{F}_s &= \{f : \mathcal{L} \rightarrow \mathcal{P}(S) \mid \forall a \in \mathcal{L} : f(a) \in C^a_{\{s\}} \vee (a \notin \mathbf{O}_{\mapsto}(s) \wedge f(a) = \emptyset)\} \\
S' &= \{s_{\text{all}}\} \cup \{(s, x) \mid s \in S \wedge x \in \mathrm{F}_s \cup \{-\} \cup \mathcal{P}(S)\} \\
{S^0}' &= \{(s, x) \mid s \in S^0 \wedge x \in \mathrm{F}_s\} \qquad\qquad W^a_s = (\{s\} \circ \overset{a}{\mapsto}) \cup \{\{s\} \circ \xrightarrow{a}\} \\
\rightarrow' &= \hookrightarrow' \cup (\{s_{\text{all}}\} \times \mathcal{L} \times \{s_{\text{all}}\}) \cup \{((s, x), a, s_{\text{all}}) \mid x \in \{-\} \cup \mathcal{P}(S) \setminus W^a_s\} \cup \\
&\quad\ \{((s, \ddot{S}), a, (s', f')) \mid s' \in \ddot{S} \wedge \ddot{S} \in W^a_s \wedge f' \in \mathrm{F}_{s'}\} \\
\hookrightarrow' &= \{((s, f), a, (s', x')) \mid f \in \mathrm{F}_s \wedge s' \in f(a) \wedge x' \in \{-\} \cup \mathcal{P}(S)\} \cup \\
&\quad\ \{((s, -), a, s_{\text{all}}) \mid a \in \mathbf{O}_{\mapsto}(s)\}
\end{aligned}$$

Here, we allow the initial state set to be empty; this does not affect expressiveness, since, e.g., the modal TS  *also describes the empty language.*

This concludes our presentation of transformations since all remaining transformations can be obtained by composition, yielding quite competitive (efficient) equivalent models.

Consistency checking and expressiveness hierarchy. As a corollary, our transformations also yield a technique for checking consistency, i.e., whether the language

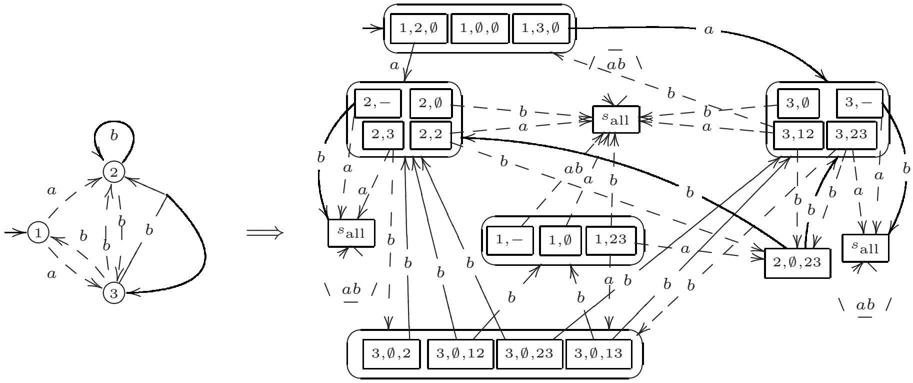
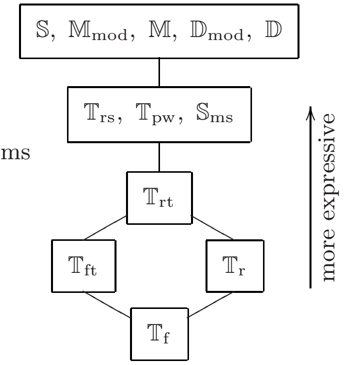


Fig. 9. Example of the transformation from $\mathbb{D}$ to $\mathbb{M}_{\text{mod}}$. On the right, the may-transitions that are implied by must-transitions are omitted, and the symbols of the state names correspond to the encoding of the transformation: (i) pairs with second element "–" correspond to the states that encode the existent labels; (ii) the remaining pairs, which have a subset as second element, encode the may-transition targets and the must-hypertransitions (states $(s, \ddot{S})$ are omitted if $\ddot{S} \notin W_s^a \cup W_s^b$, since they do not influence refinement); (iii) a triple corresponds to states that have a complete resolution where the second (resp. third) component encodes the image of a (resp. b). To improve readability, several copies of state s_{all} are used.

Considered refinement settings:

$\mathbb{D}$: Disjunctive mixed transition systems
$\mathbb{D}_{\text{mod}}$: Disjunctive modal transition systems
$\mathbb{M}$: Mixed transition systems
$\mathbb{M}_{\text{mod}}$: Modal transition systems
$\mathbb{S}$: Synchronously-communicating transition systems
$\mathbb{S}_{\text{ms}}$: Must-saturated s.-c. transition systems
$\mathbb{T}_{\text{pw}}$: Possible worlds inclusion
$\mathbb{T}_{\text{rs}}$: Ready simulation
$\mathbb{T}_{\text{rt}}$: Ready trace inclusion
$\mathbb{T}_{\text{r}}$: Ready pair inclusion
$\mathbb{T}_{\text{ft}}$: Failure trace inclusion
$\mathbb{T}_{\text{f}}$: Failure pair inclusion



$\mathbb{S}$, $\mathbb{M}_{\text{mod}}$, $\mathbb{M}$, $\mathbb{D}_{\text{mod}}$, $\mathbb{D}$
$\mathbb{T}_{\text{rs}}$, $\mathbb{T}_{\text{pw}}$, $\mathbb{S}_{\text{ms}}$
$\mathbb{T}_{\text{rt}}$
$\mathbb{T}_{\text{ft}}$
$\mathbb{T}_{\text{r}}$
$\mathbb{T}_{\text{f}}$
more expressive

Fig. 10. Refinement settings for DetTSs ordered with respect to their expressiveness

of a model is non-empty. This is trivial for trace-like settings, ready simulation settings and must-saturated STSs, because these settings cannot describe the empty language.

Corollary 1. *For a disjunctive mixed (resp. disjunctive modal, mixed, modal, synchronously-communicating) TS, consistency can be checked by transforming it via our transformations into an STS, applying the algorithm given in Prop. 2, and finally checking if the initial state set is non-empty.*

Corollary 2. *Our transformations yield the expressiveness hierarchy of refinement settings depicted in Fig. 10.*

5 Related Work

For trace and tree languages, many transformations have been developed between automata having different fairness constraints (Büchi, Muller, Rabin, Streett, parity), see, e.g., [19] and the references therein. Transformations between non-automata settings are given in [5] and [24], where the must-testing and ready simulation ($\frac{2}{3}$-bisimulation) preorders, respectively, are transformed to prebisimulation. In [16], forward/backward simulation and trace inclusion are transformed to disjunctive modal TSs (*underspecified* TSs). These transformations implicitly demonstrate that the transformed settings are less or equally expressive with respect to the describable sets of (not necessarily deterministic) TSs. Transformations preserving the complete preorder (and not only the languages) are given in [15] and [18], where [15] proves that disjunctive modal TSs can be transformed into 1-selecting modal TSs but not vice versa, whereas [18] presents transformations between modal TS variants with transition labels and predicates on states.

An alternative approach to the examination of expressiveness is taken, e.g., in [11,17], where preorders are compared regarding their coarseness. This comparison approach obviously does not lead to applicable transformations between settings.The obtained hierarchy – known as the linear-time branching-time spectrum – coincides with ours for many TS-based settings, but not in general, as is illustrated by possible worlds semantics and ready simulation semantics: by definition of possible-worlds semantics, they trivially have the same expressiveness in our language-based sense although the possible worlds preorder is finer than the ready simulation preorder. For the coinciding settings, our results cannot be immediately derived from the corresponding results in [17]. Consider, e.g., the increase of expressiveness between ready trace inclusion and ready simulation. It cannot be derived from Counterexample 8 of [17], illustrated here in Figs. 2(d) and (e), since both systems have exactly the same sets of implementations, both with respect to ready trace inclusion and ready simulation.

Yet another approach to studying the expressiveness of refinement settings is via modal logics in the style of Hennessy-Milner [20]. While much work focuses on characterizing preorders on general TSs, [3] shows a correspondence between the preorder underlying modal TSs and the prime and consistent formulas of Hennessy-Milner logic.

The problem of consistency checking is considered, e.g., in [21]. The authors present an algorithm, along with a complexity study, for checking the consistency of sets of modal TSs, i.e., for checking non-emptiness of the intersection of the modal TSs' implementation sets in terms of general TSs. [23] gives algorithms and complexity results of consistency checks for several refinement notions.

Further refinement settings have been proposed in the literature, albeit for general TSs rather than DetTSs, e.g., in [22,8,9,30,13,14]. We believe that all of them (when ignoring their possible fairness constraints) can be transformed into disjunctive mixed/modal TSs and vice versa while preserving their languages in terms of general TSs.

6 Conclusions

This paper studied the expressiveness of popular TS-based specification formalisms with respect to their describable languages in terms of deterministic TSs. Our results are summarized in the expressiveness hierarchy depicted in Fig. 10. Our work is of importance for system designers and verification-tool builders alike. The established expressiveness hierarchy aids system designers in selecting the right specification formalism for a problem in hand, while our transformations allow tool builders to reuse refinement checking algorithms across different formalisms. The results of this paper reveal that STSs combine expressiveness with succinctness and easy-to-comprehend models and thus seem to be a good choice for modeling sets of DetTSs.

Regarding future work, we wish to examine the succinctness of our refinement settings, show that our transformations lie in optimal complexity classes, and compare refinement settings based on preorders that abstract from internal computation, e.g., those mentioned in [23].

References

1. Baeten, J., Bergstra, J., Klop, J.: Ready-trace semantics for concrete process algebra with the priority operator. Computer J. 30(6), 498–506 (1987)
2. Bloom, B., Istrail, S., Meyer, A.: Bisimulation can't be traced. J. ACM 42(1), 232–268 (1995)
3. Boudol, G., Larsen, K.G.: Graphical vs. logical specifications. Theoretical Computer Science 106(1), 3–20 (1992)
4. Brookes, S., Hoare, C., Roscoe, A.: A theory of communicating sequential processes. J. ACM 31(3), 560–599 (1984)
5. Cleaveland, R., Hennessy, M.: Testing equivalence as a bisimulation equivalence. Formal Aspects of Computing 5(1), 1–20 (1993)
6. Cleaveland, R., Sokolsky, O.: Equivalence and preorder checking for finite-state systems. In: Handbook of Process Algebra, pp. 391–424. North-Holland, Amsterdam (2001)
7. Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. ACM TOPLAS 19(2), 253–291 (1997)
8. Dams, D., Namjoshi, K.S.: The existence of finite abstractions for branching time model checking. In: LICS, pp. 335–344. IEEE, Los Alamitos (2004)
9. Dams, D., Namjoshi, K.S.: Automata as abstractions. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 216–232. Springer, Heidelberg (2005)
10. Engelfriet, J.: Determinacy → (observation equivalence = trace equivalence). Theoretical Computer Science 36, 21–25 (1985)
11. Eshuis, R., Fokkinga, M.M.: Comparing refinements for failure and bisimulation semantics. Fundam. Inf. 52(4), 297–321 (2002)

12. Fecher, H., Grabe, I.: Finite abstract models for deterministic transition systems: Fair parallel composition and refinement-preserving logic. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 1–16. Springer, Heidelberg (2007)
13. Fecher, H., Huth, M.: Ranked predicate abstraction for branching time: Complete, incremental, and precise. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 322–336. Springer, Heidelberg (2006)
14. Fecher, H., Huth, M., Schmidt, H., Schönborn, J.: Refinement sensitive formal semantics of state machines with persistent choice. In: AVoCS. ENTCS (2007) (to appear)
15. Fecher, H., Schmidt, H.: Comparing disjunctive modal transition systems with an one-selecting variant. J. Logic and Algebraic Programming 77, 20–39 (2008)
16. Fecher, H., Steffen, M.: Characteristic μ-calculus formula for an underspecified transition system. In: EXPRESS 2004. ENTCS, vol. 128, pp. 103–116 (2005)
17. Glabbeek, R.v.: The linear time–branching time spectrum I. The semantics of concrete, sequential processes. In: Handbook of Process Algebra, pp. 3–99. North-Holland, Amsterdam (2001)
18. Godefroid, P., Jagadeesan, R.: On the expressiveness of 3-valued models. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 206–222. Springer, Heidelberg (2002)
19. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games. LNCS, vol. 2500. Springer, Heidelberg (2002)
20. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. J. ACM 32(1), 137–161 (1985)
21. Hussain, A., Huth, M.: Automata games for multiple-model checking. ENTCS 155, 401–421 (2006)
22. Janin, D., Walukiewicz, I.: Automata for the modal mu-calculus and related results. In: Hájek, P., Wiedermann, J. (eds.) MFCS 1995. LNCS, vol. 969, pp. 552–562. Springer, Heidelberg (1995)
23. Larsen, K.G., Nyman, U., Wasowski, A.: On modal refinement and consistency. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 105–119. Springer, Heidelberg (2007)
24. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Inf. Comput. 94(1), 1–28 (1991)
25. Larsen, K.G., Thomsen, B.: A modal process logic. In: LICS, pp. 203–210. IEEE, Los Alamitos (1988)
26. Larsen, K.G., Xinxin, L.: Equation solving using modal transition systems. In: LICS, pp. 108–117. IEEE, Los Alamitos (1990)
27. Olderog, E., Hoare, C.: Specification-oriented semantics for communicating processes. Acta Informatica 23(1), 9–66 (1986)
28. Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981)
29. Phillips, I.: Refusal testing. Theoretical Computer Science 50(3), 241–284 (1987)
30. Shoham, S., Grumberg, O.: 3-valued abstraction: More precision at less cost. In: LICS, pp. 399–410. IEEE, Los Alamitos (2006)
31. Shukla, S., Hunt, H., Rosenkrantz, D., Stearns, R.: On the complexity of relational problems for finite state processes. In: Meyer auf der Heide, F., Monien, B. (eds.) ICALP 1996. LNCS, vol. 1099, pp. 466–477. Springer, Heidelberg (1996)
32. Veglioni, S., de Nicola, R.: Possible worlds for process algebras. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 179–193. Springer, Heidelberg (1998)

Bounded Rational Search for On-the-Fly Model Checking of LTL Properties

Razieh Behjati[1], Marjan Sirjani[1,2,3], and Majid Nili Ahmadabadi[1,4]

[1] Department of Electrical and Computer Engineering,
University of Tehran, Tehran, Iran
[2] School of Computer Science, Reykjavik University, Iceland
[3] School of Computer Science,
Institute for Research in Fundamental Sciences, Tehran, Iran
[4] School of Cognitive Science,
Institute for Research in Fundamental Sciences, Tehran, Iran
behjati@ut.ac.ir, msirjani@ut.ac.ir, mnili@ut.ac.ir

Abstract. Model checking is considered as a promising approach for assuring the reliability of concurrent systems. Besides its strength it suffers from the state explosion problem, which reduces its applicability especially when systems grow larger. In this paper we propose a bounded rational verification approach for on-the-fly model checking of LTL properties. We optimize memory usage by increasing the probability of finding counter-examples. Since in on-the-fly model checking we do not have complete knowledge about the model, we use a machine learning method based on interaction and reward receiving. Based on the concept of fairness we propose a heuristic for defining rewards. We also exploit the ideas of probabilistic model checking in order to find a measure of correctness of the system in the case where no violations are found after generating a certain number of runs of the system. The experimental results show that this approach easily outperforms classic model checking approaches.

Keywords: Concurrent Systems Verification, Reinforcement Learning, Approximate Probabilistic Model Checking.

1 Introduction

Model checking [5,7] is the problem of deciding whether or not a property specified in temporal logic holds for a system. Recently, it has gained wide acceptance within the hardware and protocol verification communities, and is finding more applications in the domain of software verification. The great feature of this technique is that it can be done fully automatically, provided that the state space of the system under investigation is finite. Given a model and the desired property, an automatic model checker either returns true, meaning that the model satisfies the property, or provides a counter-example which is a possible run of the model violating the given property.

Unfortunately, model checking has its own drawbacks which make it inapplicable in many practical cases. The most important problem against model checking

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 292–307, 2010.

is the *state explosion problem.* This problem concerns the size of the system's state space which grows exponentially in the size of the model's specification. Over the past two decades, researchers have developed a large number of techniques to overcome state explosion, including: *partial-order reduction* methods [14,29], *symmetry reduction* [6,12,25], *bounded model checking* [3], and *symbolic model checking* [4,22].

In this paper we propose a technique to tackle state explosion problem. As stated by Clarke and Wing in [8] a model checking tool must be *error detection oriented.* Which means, it must be optimized for finding errors and not for guaranteeing correctness. Therefore, in our proposed technique we seek two main goals. The first goal is to make the model checking procedure more efficient in finding bugs (faulty runs) by improving the search algorithm using Reinforcement Learning. The second goal is to provide an approximate answer for the outsized models where the typical model checkers usually end up with state space explosion.

General Idea

To achieve the first goal we design and develop a framework in which a Reinforcement Learning agent walks through the state space looking for counter-examples. In the Reinforcement Learning framework an agent learns through interactions with an environment. In this framework, for each of its actions the agent receives a reward, which is the key to guide the agent to its goal. The LTL model checking is based on searching for a 'fair accepting cycle' in the Büchi automaton, B, which is the product of the automaton of the model, B_S, and the automaton of the negation of the property, $B_{\neg\varphi}$. We define a reward function which gives the agent negative rewards for finding non-accepting cycles and positive rewards for finding unfair accepting cycles. If an accepting cycle is fair then a counter example is found, the goal is reached and the search is over. We use a Monte Carlo method to control the agent's search. Monte Carlo methods are a class of computational algorithms that rely on repeated random sampling to compute their results and tend to be used when it is infeasible or impossible to compute an exact result with a deterministic algorithm. This feature allows them to provide an answer even without having the complete knowledge about the underlying problem. In this context, using a Monte Carlo method to control the agent's search allows us to benefit from the *on-the-fly* technique in LTL model checking.

In our proposed approach if a counter-example (a fair accepting cycle) is found, the model checking procedure stops and reports it. Otherwise, the agent continues searching the state space until the number of generated runs reaches an upper bound, N. To provide a termination condition for the agent's search and to provide an answer to the model checking problem even for very large systems, we use a probabilistic approach, specifically the quantitative method presented in [15], which provides an (ϵ, δ)-approximation of the correctness of the model. In this technique N is proportional to $1/\epsilon$ and $ln\ 1/\delta$, but the size of the model does not affect its value. Therefore, it provides an effective approach for verifying models with very large or even infinite state spaces.

The quantitative model checking approach as presented in [15], is applicable when the sample runs are generated independently according to an identical distribution which provides unbiased probabilities for randomly generating sample runs. But in the guided search that we use the sample runs are not generated independently. After a sample run is evaluated, the rewarding system results in a change in the probabilities, therefore making the generation of runs that violate (satisfy) the property more (less) probable. In fact, the rewarding system results in a bias in the probabilities of generating sample runs. For the reward function proposed in this paper we have proved (Appendix A) that this bias strengthens the (ϵ, δ)-approximation of [15], which enables us to use this approximation in our proposed approach.

The closest work to our approach is presented in [1], which uses Reinforcement Learning for checking liveness properties. But there is a significant difference between the two Reinforcement Learning settings. We search the product Büchi automaton B, instead of the model's Büchi automaton B_S, which allows us to model check liveness properties as well as all other LTL properties. We also consider quantitative model checking which lets us to provide a stopping criterion for the search and to use that in order to provide a probabilistic reply for oversized models.

To the best of our knowledge it is the first work in the literature which augments a learning-based, guided search method for error detection with probabilistic approaches which provides a measure of correctness when the guided search fails to find a counter-example. The main features of the proposed technique are the following:

- The technique uses a machine learning approach, or more precisely, trains a Reinforcement Learning agent for improving the search algorithm and rapidly finding counter-examples.
- It exploits the ideas of probabilistic model checking in order to provide an appropriate termination condition for the agent's search.
- Since we use a Monte Carlo method for evaluating the states we can get benefit from the on-the-fly technique in LTL model checking.

In the rest of the paper we first describe backgrounds of the work, including LTL model checking, Reinforcement Learning, and quantitative model checking. Next, in Section 3, we specify how these approaches are put together to provide an effective model checking framework. Experimental results are presented in Section 4. Section 5 briefly introduces the related works. Finally we conclude the work in Section 6, and propose directions for future works.

2 Backgrounds

2.1 LTL Model Checking

Given a concurrent system S and a linear-temporal-logic formula φ, the LTL model checking problem is to decide whether S satisfies φ. This problem can be elegantly solved by reducing it to the language emptiness problem for finite automata over infinite words [30]. For this purpose the system, S, and negation of the property, $\neg\varphi$, are modeled as Büchi automata B_S and $B_{\neg\varphi}$, respectively. The product Büchi automaton, B, is then defined as $B = B_S \times B_{\neg\varphi}$, and is used to check whether the language $L(B)$ of B is empty.

Usually we are only interested in finding counter-examples which violate φ. Therefore, our main goal is to prove non-emptiness of B, by finding a cycle that is reachable from an initial state and contains an accepting state. In addition, this accepting cycle must be fair, meaning that any enabled transitions should eventually get a turn to be executed within the cycle.

Since Büchi automaton B_S can grow exponentially large for a concurrent system S, one can avoid the state explosion problem by avoiding the explicit construction of B_S. The alternate solution is to generate the initial states of B_S first and then generate next states on demand. This *on-the-fly* approach considerably reduces the space requirements, since it constructs only the reachable part of B_S. This approach is especially useful when we are only looking for errors not proving the correctness of the system.

2.2 Reinforcement Learning and Monte Carlo Policy

Reinforcement Learning [27] refers to a learning framework for learning from interaction to achieve a goal. The learner in this framework is called the *agent* and everything outside the agent, which the agent interacts with, is called the *environment*. These interact continually; the agent selects actions and the environment responds to those actions by presenting new situations to the agent and giving it rewards. The agent's goal is to maximize values of the rewards that it gains over time.

More specifically, the agent and the environment interact in a sequence of discrete time steps, $t = 0, 1, 2, 3, \cdots$. At each time step, t, the agent receives a representation of the environment's *state*, $s_t \in S$, where S is the set of possible states, and on that basis selects an *action*, $a_t \in \mathcal{A}(t)$, where $\mathcal{A}(t)$ is the set of actions available in state s_t. One time step later, as a consequence of its action, the agent receives a numerical reward, $r_{t+1} \in \mathcal{R}$, and finds itself in a new state, s_{t+1} [27]. In many cases the agent-environment interaction breaks naturally into subsequences, which are called *episodes*. Each episode ends in a special state called the *terminal state*, followed by a reset to a starting state. Tasks with episodes of this kind are called *episodic tasks*. Fig. 1 shows the agent-environment interaction.

During the learning procedure the agent develops a *policy* denoted π_t, which is a mapping from state representations to probabilities of selecting possible actions. In general, the agent in the Reinforcement Learning problem is looking for a policy that maximizes the *expected return*, where the return R_t, is defined as some specific function of the reward sequence. Using the return values one can define the *value function* Q on states. The value of a state is the total amount of reward an agent can expect to accumulate over the future starting from that state. The value of taking action a in state s under a policy π, denoted

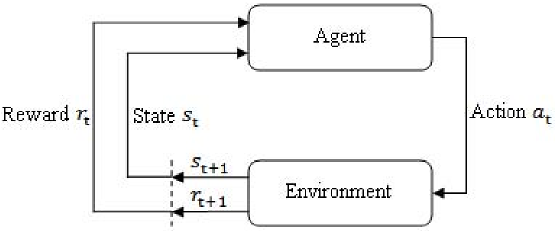


Fig. 1. The Reinforcement Learning framework

$Q^\pi(s,a)$, is defined as the expected return starting from s, taking the action a, and thereafter following policy π:

$$Q^\pi(s,a) = E_\pi\{R_t|s_t = s, a_t = a\}.$$

There are several ways for defining the return using the reward sequence. One approach is to invoke the concept of *discounting*. According to this approach, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized. In particular, it chooses a_t to maximize the expected discounted return:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1};$$

where $0 \leq \gamma \leq 1$ is called the *discount rate.*

Several approaches exist for determining the optimal policy. One of these approaches is the *Monte Carlo method*, which is the method that we used in our work. In this approach, the value function is repeatedly altered to more closely approximate the value function for the current policy, and the policy is repeatedly improved with respect to the current value function [27].

2.3 Quantitative Model Checking

In this section we briefly introduce quantitative model checking [15], in which a Monte Carlo method is used to compute an (ϵ, δ)-approximation of the correctness of a model. The main idea in quantitative model checking is to use N independent random samples $Z_1, \ldots, Z_n$ identically distributed according to random variable Z with mean μ_Z, and to take $\tilde{\mu}_Z = (Z_1 + \ldots + Z_N)/N$ as the approximation of μ_Z [15]. In the model checking problem each Z_i, represents a sample run in the model. For an accepting run the value of Z_i is set to zero and for a non-accepting run it is set to one. In this setting μ_Z represents the probability of encountering a non-accepting run in the model.

An important issue in providing an (ϵ, δ)-approximation is determining the value of N. According to the *zero-one estimator theorem* [20], if N is proportional to $\Upsilon = 4ln(2/\delta)/\mu_Z\epsilon^2$ then $\tilde{\mu}_Z$ approximates μ_Z with absolute error ϵ and with probability $1 - \delta$; in other words $Pr[\mu_Z(1-\epsilon) \leq \tilde{\mu}_Z \leq \mu_Z(1+\epsilon)] \geq 1 - \delta$.

There are two main difficulties in using the zero-one estimator for the calculation of N. The first is that N depends on $1/\mu_Z$, the inverse of the value that we want to approximate. The second difficulty comes from the factor $1/\mu_Z\epsilon^2$ in the expression for Υ, which can result in unnecessarily large values for N. To solve these problems, [9] has proposed *optimal approximation algorithm* (OAA) by introducing a more practical approach for calculating N, called *generalized zero-one estimator theorem.* The OAA algorithm makes use of the outcomes of previous experiments to compute N. We also use this algorithm to find the appropriate value for N.

3 The Verification Procedure

In this section, we explain our model checking algorithm. We first describe the structure and settings of the Reinforcement Learning problem. We continue the explanation about our proposed approach by describing how probabilistic methods are used to provide a termination condition for agent's search. For the termination condition we provide an upper-bound N on the number of generated paths. One should note that for small models where the paths are too short this termination condition may stop the procedure when there is still plenty of free memory available, while on the other hand for very large systems it may still encounter state explosion problem. Compared to the classical approaches, it is less probable for our proposed approach -due to its approximate and error detection nature- to encounter the state explosion problem.

3.1 Applying Reinforcement Learning

The first step to form a Reinforcement Learning solution to a problem is to define the sets of states and actions, the reward function, the value function and the agent's policy. Structural similarities between the Reinforcement Learning and the model checking problem have simplified this step.

States and Actions. For a model S and a property φ, we use $B = B_S \times B_{\neg\varphi}$ to define states and actions of the RL problem. For any state s in B there is a state, s^{RL}, in the RL problem and for any transition outgoing from s there is an action $a \in \mathcal{A}(s^{RL})$. In this task the set of initial states is fixed and the agent starts its search from one of these initial states.

Fairness. Before we can go any further in establishing the RL problem, we should note that LTL model checking only concerns fair runs, which means that among all the accepting cycles only fair ones represent a valid counter-example. There are two ways to include fairness. In the first method, introduced in [1], finding fair paths is considered as part of the agent's goal. Therefore, the agent's task is to look for fair, accepting paths. To achieve this, the states of the RL problem should also contain information about the executed processes and their order of execution, which results in an increase both in the number of states and the size of states, leading to a much larger state space.

To avoid unnecessarily large numbers of states in our approach, we use a second way for ensuring fairness of the reported counter-examples. Here, we let the agent to generate both fair and unfair paths. Therefore, if the agent reports an accepting cycle, its fairness should be checked first. If the cycle is fair, it is a valid counter-example. Otherwise, the agent receives an appropriate reward and continues to searching for the next run.

Reward Function. In the RL framework the agent is directed to its goal by the means of a reward function. Here the goal is to find a counter-example. In fact the agent is looking for an accepting cycle in the Büchi automaton B, which is a path ending to a loop with at least one accepting state in that loop. Therefore, the agent's task is to start from an initial state, then take a sequence of actions until it reaches a cycle. If the cycle is accepting and fair then the agent has found a counter-example, otherwise it should reset the state to an initial state and start looking for the next run. Since it is an episodic task the agent generates paths in order to gain the maximum return. We use the discounted return (Section 2.2) with a discounting factor γ for this task.

Therefore, the most important part of the work is to define a reward function that appropriately provides a discounted return which correctly leads the agent to its goal. For this purpose we assign a negative reward to a transition (action) leading to a recurring state if it is at the end of a non-accepting cycle, a positive reward if it is at the end of an unfair accepting cycle, and zero to all other transitions. Note that if the accepting cycle which the agent finds is a fair cycle then a valid counter-example is found and we are done. Using this heuristic, we discourage the agent from seeking non-accepting cycles by giving it negative rewards any time it finds one, and we encourage it to find accepting cycles by positively rewarding finding such cycles.

Value Function and Policy. To completely determine the task we must also establish the value function and the agent's policy. We use a Monte Carlo method for this purpose. Monte Carlo methods are ways of solving the Reinforcement Learning problem based on averaging sample returns. The most important feature of Monte Carlo methods is that they do not assume complete knowledge of the environment. Monte Carlo methods require only experience sample sequences of states, actions, and rewards from actual or simulated interaction with an environment [27]. This setting updates π, the agent's policy, and Q, the value function, according to the following formulas:

$$\pi(s) = arg\ max_a\ Q(s,\ a) \tag{1}$$

$$Q^{\pi_k}(s, \pi_{k+1}(s)) = Q^{\pi_k}(s, arg\ max_a\ Q^{\pi_k}(s,\ a)) = max_a\ Q^{\pi_k}(s,\ a) \tag{2}$$

We use a *soft-max*, *on-policy* algorithm to iteratively evaluate π and Q. The soft-max nature of the algorithm ensures that all possible state-action pairs has the chance of being selected by the agent. The algorithm is in fact *ε-greedy*, meaning that it assigns the largest probability to the best action, but do not completely eliminate other actions, allowing the agent to discover even better actions in the future. For this purpose the algorithm starts by setting ϵ to one, then reduces it (for example by dividing it by two) after generating every K episodes.

On-the-fly Model Checking. Using a Monte Carlo method which does not assume complete knowledge of the environment allows us to get benefit from on-the-fly model checking. For this purpose we input a description of the desired model, and the Büchi automaton for the negation of the property. While making an episode we use the model's description and the property to generate new states from the previous ones. Since we do not store the state space, we may do this over and over for a special pair of source and target states. Since it is a low cost task it does not harm the performance, while on the other hand it allows us to save memory.

3.2 Providing a Measure of Correctness

We use the idea of quantitative model checking to provide an upper bound, N, on the number of episodes that the agent must generate before it stops searching the state space. If the agent finds an accepting cycle it reports that as a counterexample, otherwise it should continue generating episodes until it generates N non-accepting cycles. N is the optimum upper bound on the number of episodes according to which p_Z, the probability of the correctness of the model, satisfies $Pr[p_Z \geq 1/(1+\epsilon)] \geq 1-\delta$. We use OAA to calculate N. The inputs to OAA are a number of random variables with mean p_Z, each of which indicating a run of the model (an episode). The value of a random variable representing an accepting run is one, and for a non-accepting run it is zero. Since we are only interested in the situations where all the runs are non-accepting, we assume that all inputs of OAA are equal to 1, which yields in a the following formula for the value of N:

$$N = \epsilon \times 2(1+\sqrt{\epsilon})(1+2\sqrt{\epsilon})(1+\frac{ln\ 3/2}{ln\ 2/\delta})\gamma$$

$$\gamma = \frac{4(e-2)ln(2/\delta)}{\epsilon^2}$$

We can always increase the value of N such that the confidence and probability get higher and higher, while state explosion has not stopped us.

The main difference between our proposed approach and the quantitative model checking proposed in [15] comes from the difference in assigning probabilities to transitions. In the quantitative model checking, all transitions outgoing from a specific state have equal probabilities, while in our approach the RL agent assigns different probabilities to such transitions. In a state s, the agent takes the transition that it has found to be the best transition with probability $1-\xi+\frac{\xi}{|\mathcal{A}(s)|}$, and takes each of the other transitions with probability $\frac{\xi}{|\mathcal{A}(s)|}$.

Another difference is that, probabilities that the agent assigns to transitions change over time as it searches more, while in the quantitative model checking

```
Learning Based Probabilistic Model Checking Algorithm
input: the specification of the model (S), and the desired property ($\phi$)
input: $0 < \epsilon,\ \delta \leq 1$
output: either (false, counter-example) or (true, $Pr[p_z \geq 1/(1+\epsilon)] \geq\ 1-\delta$)
(1) Calculate $N$ using $\epsilon$ and $\delta$

(2) Initialize for all $s \in S$, and $a \in \mathcal{A}(s)$:
      $Q(s,a) \leftarrow$ arbitrary
      $Returns(s,a) \leftarrow$ empty list
      $\pi \leftarrow$ an arbitrary soft-max policy

(3) Repeat for $N$ times:
      (a) Generate an episode using $\pi$
      (b) If the episode represents an accepting run and the run is fair
            return (false, the run);
      (c) Else
            For each pair $s$, $a$ appearing in the episode
                $R \leftarrow$ return following the first occurrence of $s$, $a$
                Append $R$ to $Returns(s,a)$
                $Q(s,a)\ \leftarrow\ average(Returns(s,a))$
            For each $s$ in the episode
                $a^* \leftarrow\ arg\ max_a\ Q(s,a)$
                For all $a \in \mathcal{A}(s)$:
                    $\pi(s,a)\ \leftarrow \begin{cases} 1-\xi+\frac{\xi}{|\mathcal{A}|}, & a\ =\ a^*; \\ \frac{\xi}{|\mathcal{A}|}, & a\ \neq\ a^*. \end{cases}$

(4) return (true, $Pr[p_z \geq 1/(1+\epsilon)] \geq\ 1-\delta$)
```

Fig. 2. The proposed probabilistic model checking algorithm

the probabilities are fixed. In fact the agent starts with equal probabilities for all transition outgoing from a state, then it changes these probabilities in order to make transitions leading to counter-examples more probable. In other words, it starts from an unbiased setting then makes a bias towards finding accepting runs. Since this bias strengthens $Pr[p_z \geq 1/(1+\epsilon)] \geq\ 1-\delta$, we can still use this method to approximate the correctness of the model.

The key to gain the desired bias in the probabilities is the correct definition of the reward function. For a reward function that assigns a negative reward (e.g. -1) for non-accepting cycles and a positive reward (e.g. 1) for unfair accepting cycles, this property holds (the proof is available in Appendix A). Fig 2 shows our proposed model checking algorithm.

4 Experimental Results

To evaluate the feasibility and performance of our proposed algorithm we implemented and used it to model check the dining philosophers problem for different sizes. For the modeling language we used Rebeca [24], which is an actor-based language for modeling concurrent, reactive systems.

A Rebeca model consists of a set of rebecs (reactive objects) which are concurrently executed. Rebecs are encapsulated active objects, with no shared variables. Each rebec is instantiated from a Reactive-Class and has a single thread of execution which is triggered by reading messages from an unbounded message queue. Rebecs communicate by asynchronous message passing. The messages which are sent by the sender rebec are put in the message queue of the receiver rebec. The receiver takes a message from top of its message queue and executes the corresponding message server atomically.

Dining philosophers problem: The model for the dining philosophers problem is simple. n philosophers are sitting around a table with n forks among them. A philosopher either thinks or picks up the two forks next to him and eats. After finishing his meal, he releases the forks and continues thinking.

There are two reactive-classes in the Rebeca model that we used in our experiments, one for representing philosophers and the other for representing forks. In this model a philosopher requests both his forks at the same time, since message passing is asynchronous in Rebeca, the left and right forks may serve these requests in any order. The philosopher then waits until it acquires both forks, then releases both of them. Again these messages may be served in any order.

We have model checked two properties for this model. The first one is a safety property ensuring that a specific fork is not kept by two philosophers simultaneously. The given model satisfies this property. Model checking results for this property are given in Table 1. We examined the approach several times and for each value of n, three sample trials are reported in Table 1. As shown in the first column of the table, in this experiment ϵ was set to 0.053, and δ was set to 0.1, therefore after generating $N = 662$ correct runs the algorithm reports with confidence more than 90% that the model is correct with probability more than 95%. The column labeled 'Path Len.' determines the maximum and average lengthes of the paths generated in each trial. The total number of states generated in each trial is also given in the sixth column.

We used Modere [19], the Rebeca model checking engine, to verify the same model. This time we were stopped at $n = 5$, because of the state explosion problem. The results are shown in Table 3. Compared to Modere which is a classical model checker, our approach is very efficient and capable of checking models with a very large state space. The results also show that considering time and space, for small models (e.g. $n = 2$ in dining philosophers problem), the classical approaches are better choices while for larger models ($n \geq 4$), where classical approaches fail, our approach is very promising.

The next property, which is a liveness property, ensures that whenever a philosopher acquires one of his forks, he eventually eats. This property is not satisfied in the model. The results for this experiment are shown in Table 2. This table has an additional column labeled by '# paths' which determines the number of paths generated before reaching the counter example. Again we used Modere to verify the model for the same property. The results are shown

Table 1. Model checking results for the safety property, using the proposed approach. The last column is the maximum total memory the program used during its execution.

ϵ, δ, N	n	**Trial**	**Path Len.** (Avg/Max)	**Time** (ms)	**# States**	**Mem** (MB)
0.053, 0.1, 662	2	1	34.95/66	4579	124	64.92
		2	35.18/90	4594	130	64.85
		3	35.42/74	4657	127	64.5
	4	1	77.09/492	14813	3604	69.43
		2	78.56/502	15359	3627	75.61
		3	77.06/415	14672	3550	81.75
	5	1	112.39/498	23938	13080	73.23
		2	111.21/483	23921	12512	72.78
		3	112.00/572	25655	13312	82.46
	10	1	1647.11/35936	104937688	324542	724.3
		2	1300.68/30119	64579203	227908	490.27
		3	1457.09/27812	82241000	247318	616.17

Table 2. Model checking results for the liveness property, using the proposed approach. The last column is the maximum total memory the program used during its execution.

ϵ, δ, N	n	**Trial**	**Path Len.** (Avg/Max)	**Time** (ms)	**# paths**	**# States**	**Mem** (MB)
0.053, 0.1, 662	2	1	33.25/42	16	4	62	5.04
		2	35.21/58	234	14	61	8.82
		3	36.3/61	15	3	70	21.4
	4	1	86.92/172	297	14	1170	60.48
		2	49.2/50	47	5	173	37.56
		3	75.4/90	78	5	234	58.12
	5	1	154.17/374	281	6	845	51.07
		2	112.44/345	563	18	200	58.08
		3	97.11/234	515	19	195	62.48
	10	1	829.43/34195	10690328	432	94761	699.59
		2	923.28/27005	10859750	411	72438	218.85
		3	1570.34/196857	31652797	317	201944	711.64

Table 3. Model checking results using Modere

	Safety Property			**Liveness Property**		
n	**Time**	**Path Len. (Max)**	**# States**	**Time**	**Path Len. (Max)**	**# States**
2	< 1s	117	1266	< 1s	136	1375
4	> 33h	not available	not available	≈ 3s	1674	32747

Table 4. Comparison of the average number of runs generated before finding a counter-example

n	**Random model checking**	**The proposed approach**
4	44.33	7.17
5	66.33	12.33

in Table 3. According to the results for $n = 2$ and $n = 4$, our approach is strongly optimized for finding counter-examples, since it is able to find such paths after generating a less number of states compared to Modere. Comparing the lengthes, our approach also finds shorter paths. We can conclude that our approach outperforms Modere also in finding counter-examples.

In order to provide a better evaluation of our approach we implemented a random model checker and used it to verify the liveness property of the dining philosophers problem for $n = 4$ and $n = 5$. Table 4 shows the average number of runs generated by each of the tools before finding a counter-example. The results show that our proposed approach is more efficient than random model checking since it can find counter-examples after generating a less number of runs.

5 Related Works

Related works fall into two categories. The first category includes works that use Reinforcement Learning to find counter-examples in a model. The only work in this category is presented in [1]. There are two main differences between our approach and the one presented in [1]. First, in [1] the reward function is defined in a way to find cycles in B_S that violate a given liveness property, without involving $B_{\neg\varphi}$ in the state space of the Reinforcement Learning problem. In our approach on the other hand we have set up the Reinforcement Learning problem based on the product $B = B_S \times B_{\neg\varphi}$, which allows us to model check all LTL properties. The second difference regards the termination condition. In [1], it is not clear what does the agent do when the model is infinite, or when no counter-example is found before encountering state explosion. As specified earlier we use the idea of quantitative model checking to overcome this problem.

Probabilistic approaches to the model checking problem offer another category of related works. A group of these works concern stochastic model checking. These approaches are used for model checking probabilistic models, since regular model checking algorithms are not applicable to them. The works presented in [2,13,21], are samples of stochastic model checking.

In addition to these approaches there are other approaches that invoke probabilistic reasoning in model checking non-probabilistic models. Random model checking is one of these approaches. The basis of random model checking is to provide a random walk through the state space. In contrast to the classical model checking which uses Breadth First Search, in random model checking nothing is stored, only a set of random runs of the model is generated and evaluated. Random model checkers generate and check many runs, some of which may occur more than once. One can increase the probability of finding counter-examples by generating more runs. But, regardless of the number of generated paths, it is not possible to certainly conclude the model is correct if there are no counter-examples found. Therefore these approaches can only provide an approximation of the correctness of the model.

In [17] it is shown that for certain specifications the CTL model checking problem reduces to a question of reachability in the system's state-transition graph, and applies a simple, randomized algorithm to this problem. To answer this question, a random walk starts from an initial state and continues until a state with a given label is found. If such a state is found, the model checking procedure finishes, otherwise it continues until a total number of transitions are followed. At this point the algorithm reports that no state with the given label is reachable. However there would a probability of error in this answer, which is in inverse proportion to the number of explored transitions. The main limitation of this approach is that it is only applicable for a subset of CTL properties. The main advantage of this approach is that it is very efficient according to time and memory costs.

Another probabilistic algorithm for model checking is presented in [28]. This algorithm trades space with time: when the memory is over because of state explosion the algorithm does not give up verification, instead it just proceeds at a lower speed and its results will hold with some arbitrarily small probability of error. In addition, [26] proposes various heuristic algorithms that combine random walks on the state space with bounded breadth first search in a parallel context.

Quantitative model checking [15] and Monte Carlo model checking [16] are other randomized approached that use Monte Carlo methods to provide a measure of correctness of the model. In these works an $(\epsilon,\ \delta)$-approximation of the correctness of the model is calculated. Similar works are also proposed in [18,10]. In fact we obtained the idea of providing an $(\epsilon,\ \delta)$-approximation of the model's correctness from these works. These works search the state space via random walks therefore do not face the state space explosion problem. Together with the $(\epsilon,\ \delta)$-approximation this feature has made these approaches very promising. The contribution of our work to these approaches is that we have replaced the random search with a guided search which allows us to find counter-examples more rapidly.

6 Conclusion and Future Works

In this paper we proposed an efficient approach for model checking large or infinite state systems. In the design of the proposed approach we concerned the following features:

1. A model checker should be error detection oriented, meaning that it must be optimized for finding errors and not for certifying correctness.
2. A model checker should terminate with an answer, even an approximate one.

We use a Reinforcement Learning agent in order to optimize memory usage by providing a guided search for finding counter-examples. We invoke the concept of fairness to propose a heuristic reward function which correctly leads the agent to finding counter-examples. Besides, we use probabilistic methods to provide a termination condition for the agent's search as well as to provide an approximate measure for the correctness of the model. The experimental results show that our approach can go beyond the limits of classical model checkers. While classical model checkers are better choices for the verification of small models, our approach is promising for model checking large or infinite state models, especially when the model is faulty. Experimental results also show that the proposed approach outperforms random model checking.

Currently our approach is limited to LTL properties. An interesting direction to continue the work is to extend it to the model checking problem of Computational Tree Logic [11,23] formulas. In the future we should also work on providing a more accurate, tighter approximation of the correctness of the model, considering the bias in the probabilities that the agent assigns to transitions. We have a plan to improve the implementation of the proposed algorithm and add it as a special capability to Modere.

References

1. Araragi, T., Cho, S.M.: Checking liveness properties of concurrent systems by reinforcement learning. In: Edelkamp, S., Lomuscio, A. (eds.) MoChArt IV. LNCS (LNAI), vol. 4428, pp. 84–94. Springer, Heidelberg (2007)
2. Beauquier, D., Slissenko, A., Rabinovich, A.: A logic of probability with decidable model-checking. In: Bradfield, J.C. (ed.) CSL 2002 and EACSL 2002. LNCS, vol. 2471, pp. 306–321. Springer, Heidelberg (2002)
3. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
4. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^20 states and beyond. Information and Computation 98(2), 142–170 (1992)
5. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems 8, 244–263 (1986)
6. Clarke, E.M., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. Form. Methods Syst. Des. 9(1-2), 77–104 (1996)
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
8. Clarke, E.M., Wing, J.M., et al.: Formal methods: State of the art and future directions. ACM Computing Surveys 28, 626–643 (1996)
9. Dagum, P., Karp, R., Luby, M., Ross, S.: An optimal algorithm for monte carlo estimation. In: FOCS 1995, Washington, DC, USA, p. 142. IEEE Computer Society, Los Alamitos (1995)
10. Darbon, J., Lassaigne, R., Peyro, S.: Approximate probabilistic model checking for programs. In: Second IEEE International Conference on Intelligent Computer Communication and Processing, ICCP 2006 (2006)
11. Allen Emerson, E.: Temporal and modal logic, pp. 995–1072 (1990)
12. Allen Emerson, E., Prasad Sistla, A.: Symmetry and model checking. Form. Methods Syst. Des. 9(1-2), 105–131 (1996)
13. Etessami, K., Kwiatkowska, M.Z., Vardi, M.Y., Yannakakis, M.: Multi-objective model checking of markov decision processes. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 50–65. Springer, Heidelberg (2007)
14. Godefroid, P.: Using partial orders to improve automatic verification methods. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 176–185. Springer, Heidelberg (1991)
15. Grosu, R., Smolka, S.A.: Quantitative model checking. In: ISoLA (Preliminary proceedings). Technical Report, vol. TR-2004-6, pp. 165–174. Department of Computer Science, University of Cyprus (2004)

16. Grosu, R., Smolka, S.A.: Monte carlo model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 271–286. Springer, Heidelberg (2005)
17. Haslum, P.: Model checking by random walk. In: ECSEL Workshop (1999)
18. Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate probabilistic model checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 73–84. Springer, Heidelberg (2004)
19. Jaghoori, M.M., Movaghar, A., Sirjani, M.: Modere: The model-checking engine of Rebeca. In: ACM Symposium on Applied Computing - Software Verificatin Track, pp. 1810–1815 (2006)
20. Kapp, R.M., Luby, M., Madras, N.: Monte-carlo approximation algorithms for enumeration problems. J. Algorithms 10(3), 429–448 (1989)
21. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007)
22. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic, Dordrecht (1993)
23. Pnueli, A.: The temporal semantics of concurrent programs, pp. 1–20 (1979)
24. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundamenta Informaticae 63(4), 385–410 (2004)
25. Prasad Sistla, A., Emerson, E.A.: On-the-fly model checking under fairness that exploits symmetry. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 232–243. Springer, Heidelberg (1997)
26. Sivaraj, H., Sivaraj, H., Gopalakrishnan, G., Gopalakrishnan, G.: Random walk based heuristic algorithms for distributed memory model checking. In: Proc. of Parallel and Distributed Model Checking (PDMC 2003). ENTCS, vol. 89, p. 2003. Elsevier, Amsterdam (2003)
27. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge
28. Tronci, E.: A probabilistic approach to automatic verification of concurrent systems. In: Proc. Asia-Pacific Software Engineering Conference (APSEC 2001), pp. 317–324. IEEE Computer Society, Los Alamitos (2001)
29. Valmari, A.: A stubborn attack on state explosion. Form. Methods Syst. Des. 1(4), 297–322 (1992)
30. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proc. 1st Symp. on Logic in Computer Science, Cambridge, June 1986, pp. 332–344 (1986)

A Proof of Correctness

In this section we prove that our proposed reward function provides the desired bias in the probabilities assigned to the transitions. Suppose that we have the model given in Fig. 3. We use $p_a^k(s)$ to show the sum of probabilities of paths with length k, starting from s and ending to an accepting loop. Similarly we use $p_n^k(s)$ to show the sum of probabilities of paths with length k, starting from s and ending to a non-accepting loop. Therefore, for any state s_i, at any step of the learning procedure we have:

$$\sum_{k=1}^{\infty}(p_a^k(s_i) + p_n^k(s_i)) = 1.$$

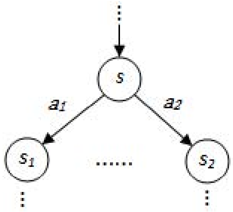


Fig. 3. A sample model

Using the discounted return defined in Section 2.2, for a reward function that assigns a negative reward of $-r_1$ to non-accepting cycles, a positive reward of r_2 to unfair accepting cycles, and no reward (zero) to all actions leading to non-terminal states, the value of an action corresponding to (s, a_i, s_i) is calculated using the following formula:

$$Q(s, a_i) = \sum_{j=1}^{\infty} \gamma^j (p_a^j(s_i) - p_n^j(s_i)), \tag{3}$$

Now we consider three possible cases:

1. **The agent has only discovered non-accepting cycles.** In this case because of the discounting factor γ^j and the negative reward assigned to such cycles, Eq. 3 results in a larger value for an action leading to a longer non-accepting path than an action leading to a shorter one. In this way this reward function conducts the agent to seek for longer paths which enables the agent to visit more states and actions and therefore makes it more probable for the agent to find accepting paths.
2. **The agent has only discovered accepting cycles.** In this case because of the discounting factor γ^j and the positive reward assigned to such cycles, Eq. 3 results in a larger value for an action leading to a shorter accepting path than an action leading to a longer one. In this way the reward function conducts the agent to find shorter counter-examples.
3. **The agent has discovered some accepting cycles and some non-accepting cycles.** In this case according to Eq. 3, the agent more probably selects transitions that ends either to longer non-accepting paths or to shorter accepting paths.

In all of the above cases the probabilities change in the direction that makes discovery of counter-examples more probable.

Automated Translation and Analysis of a ToolBus Script for Auctions

Wan Fokkink[2,1], Paul Klint[1,3], Bert Lisser[1], and Yaroslav S. Usenko[4,1]

[1] Software Engineering Cluster,
Centrum voor Wiskunde en Informatica,
Amsterdam, The Netherlands
[2] Theoretical Computer Science Section,
Vrije Universiteit Amsterdam, The Netherlands
[3] Programming Research Group,
Universiteit van Amsterdam, The Netherlands
[4] INRIA Lille - Nord Europe,
Parc Scientifique de la Haute Borne
40, avenue Halley
Bat.A, Park Plaza
59650 Villeneuve d'Ascq, France

Abstract. ToolBus allows to connect tools via a software bus. Programming is done using the scripting language Tscript, which is based on the process algebra ACP. In previous work we presented a method for analyzing a Tscript by translating it to the process algebraic language mCRL2, and then applying model checking to verify certain behavioral properties. We have implemented a prototype based on this approach. As a case study, we have applied it on a standard example from the ToolBus distribution, distributed auction, and detected a number of behavioral irregularities in this auction Tscript.

1 Introduction

ToolBus [1,2] provides a simple, service-oriented view on organizing software systems by separating the *coordination* of software components from the actual *computation* that they perform. It organizes a system along the lines of a programmable software bus. Programming is done using the scripting language Tscript that is based on the process algebra ACP (Algebra of Communicating Processes) [3] and abstract data types. The tools connected to the ToolBus can be written in any language and can run on different machines.

A Tscript can be tested, as any other software system, to observe whether it exhibits the desired behavior. An alternative approach for analyzing communication protocols is model checking, which constitutes an automated check of whether some behavioral property is satisfied. This can be, roughly, a safety property, which must be satisfied throughout any run of the system, or a liveness property, which should eventually be satisfied in any run of the system. To perform model checking, the communication protocol must be specified in some

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 308–323, 2010.

formal language, and the behavioral properties in some temporal logic. Strong points of model checking are that it attempts to perform an exhaustive exploration of the state space of a system, and that it can often be fully automated.

As one of the main aims of Tscript, Bergstra and Klint [2] mention that it should have "a formal basis and can be formally analyzed". The formal basis is offered by the process algebra ACP, but ways to formally analyze Tscripts were lacking until recently [4]. There a number of obstructions for an automatic translation from Tscript to ACP were classified, and solutions were proposed. Firstly, each Tscript process has a built-in queue to store incoming messages, which is left implicit in the process description; in mCRL2, all of these queues are specified explicitly as a separate process. Secondly, Tscript supports dynamic process creation; in mCRL2, we chose to start with a fixed number of ToolBus processes, and let a master process divide connecting tools over these processes. Thirdly, we expressed the iterative star operator of Tscript as a recursive equation in mCRL2. And fourthly, we developed some guidelines on how to deal with so-called result variables in Tscript.

The work in [4] was initiated by the developers of the ToolBus, who are keen to integrate model checking into the design process. Based on [4], we have now implemented a prototype translation from Tscript into the formal modeling language mCRL2 [5]. This language is also based on the process algebra ACP, extended with equational abstract data types [6]. As a result, Tscript can then be model checked using the mCRL2 or CADP toolset [7].

We report on an exploratory case study, to investigate in how far the automated translation from Tscript to mCRL2 can serve as a way to formally verify Tscripts. The case study concerns a distributed auction, in which the auction master and the bidders are cooperating from different computers. This auction Tscript has been used extensively for teaching purposes at various universities and in numerous demonstrations of the ToolBus. We translated the Tscript of the auction system to mCRL2, and analyzed the resulting model with several different approaches. We performed on-the-fly model checking with CADP. On-the-fly means that only the part of the state space needed for checking a property is generated; this is essential here, because the state space of the translated auction system is infinite. Moreover, we enriched the model with behavior from the environment, containing an error action that is triggered if a certain series of events occurs. To perform symbolic model checking, we translated the model and the property that we wanted to check into a Parametrized Boolean Equation System [8], and analyzed this symbolic object with the mCRL2 toolset.

This analysis revealed two deadlocks and a race condition in the auction system. First of all, a deadlock occurs when the master process is busy with a sale, and a new bidder connects to the system, but disconnects very quickly. In this case, the master wishes to synchronize with the bidder process, and will wait indefinitely for this synchronization. A second deadlock occurs since processes that subscribe to certain types of notes in the ToolBus, may never unsubscribe. This can happen when a process terminates after completing its task. Although not considered to be a real error, it does show up in our analysis. A third and more serious error is that a bidder can, for a very short time slot, bid for the last item that has already been sold, while the master interprets this as a bid for the next item. Finally, we discuss a possible Denial of Service attack. We proposed fixes for the problems we found, and verified that with these fixes the system behaves correctly.

This paper is set up as follows. Section 2 gives a brief overview (taken from [4]) of the ToolBus and Tscript, and presents the auction example. Section 3 gives a brief overview (taken from [4]) of mCRL2 and CADP. Section 4 discusses the translation scheme from Tscript to mCRL2 that originates from [4]. Section 5 presents an analysis of the auction example using this translation scheme. Finally, Section 6 contains conclusions.

Related Work. Our work has its origins in the formal verification of interface languages [9,10]. The aim is to get a separation of concerns, in which the (in our case Tscript) interfaces that connect software components can be analyzed separately from the components themselves. Our work is closest in spirit to Pipa [11], an interface specification language for an aspect-oriented extension of Java called AspectJ [12]. In [11] it is discussed how one could transform an AspectJ program together with its Pipa specification into a Java program and JML specification, in order to apply existing JML-based tools for verifying AspectJ programs, see also [13].

Many publications on model checking and other verification experiments in process algebra present one of the following two setups. Either verification of a (hand-made) *model* is presented, without an implementation in mind, or a model is *reverse engineered* from the source code of a working system, and then analyzed. Here we mention the works that focus on using process algebra for both the (*forward*) development and the analysis of a system.

ToolBus is not the only system where process algebra is used as a scripting language to describe coordination of software components. Many of these put focus on the architectural design as well as on obtaining the working executable system by either code generation or interpretation of process algebra. Some of these make use of the verification possibilities process algebra-based tools like FDR2, CADP, μCRL, mCRL2 and CWB offer.

In [14] a distributed Java system based on CSP is proposed. In [15] a methodology for control system implementation is proposed based on the ideas of [14]. In [16] Analytical Software Design (ASD) method based on Sequence-Based Specifications (SBS) [17] is presented. As demonstrated in [18], the method allows for verified software development where a CSP model is generated from SBSs and verified in FDR2. Yet another CSP-based approach is CSP++ [19], which is a C++ library for executing CSP models.

The most recent version of CAESAR from the CADP toolset provides a functionality called EXEC/CAESAR for C code generation. This C code interfaces with the real world, and can be embedded in applications. This allows rapid prototyping directly from the LOTOS specification. The implementation of the process algebraic formalism χ [20], for modeling and analyzing the dynamics and control of, for instance, production plants, is also centered around the ToolBus.

In [21] a method of software integration based on χ is presented. It allows to generate source code and test cases from χ models.

As related work in the context of the ToolBus, Diertens [22,23] uses the ToolBus to implement a platform for simulation and animation of process algebra specifications in the language PSF. In this approach, Tscript is automatically generated from a PSF specification.

2 ToolBus and Tscript

The behavior of the ToolBus consists of the parallel composition of a variable number of processes. In addition to these processes, a variable number of external tools written in different languages may be connected to the ToolBus via network sockets or OS level pipes. All interactions between processes and connected tools are controlled by Tscripts, which are based on predefined communication primitives. The classical procedure interface (a named procedure with typed arguments and a typed result) is thus replaced by a more general behavior description.

A Tscript process is built from the standard process algebraic constructs: atomic actions (including the deadlock `delta` and the internal action `tau`), alternative composition +, sequential composition · and parallel composition $\|$. The binary star operation $p * q$ represents zero or more repetitions of p, followed by q. Atomic actions are parametrized with data parameters (see below), and can be provided with a relative or absolute time stamp. A process definition is of the form $\mathit{Pname}(x_1, \ldots, x_n)$ `is` P, with P a Tscript process expression and $x_1, \ldots, x_n$ a list of data parameters. Process instances may be created dynamically using the `create` statement.

The following communication primitives are available. A process can send a message (using `snd-msg`), which should be received, synchronously, by another process (using `rec-msg`). Furthermore, a process can send a note (using `snd-note`), which is broadcast to other, interested, processes. A process may `subscribe` and `unsubscribe` to certain notes. The receiving processes read notes asynchronously (using `rec-note`) at a low priority. Processes only receive notes to which they have subscribed. Communication between ToolBus and tools is based on handshaking communication. A process may send messages in several formats to a tool (`snd-eval`, `snd-do`, `snd-ack-event`), and can receive values (`rec-value`) and events (`rec-event`) from a tool.

The only values that can be exchanged between the ToolBus and connected tools are terms of some sort (basic data types booleans, integers, strings and lists). In these terms, two types of variables are distinguished: *value* variables whose value is used in expression, and *result* variables (written with a question mark) who get a value assigned to them as a result of an action or a process call. Manipulation of data is completely transparent, i.e., data can be received from and sent to tools, but inside ToolBus there are hardly any operations on them. ATerms [24] are used to represent data terms; ATerms support maximal subterm sharing, and use a very concise, binary format. In general, an adapter is needed for each connected tool, to adapt it to the common data representation and message protocols imposed by ToolBus.

The ToolBus was introduced in the mid-1990s for the implementation of the ASF+SDF Meta-Environment [25,26] but has been used for the implementation of various other systems as well. The source code and binaries of the ToolBus and related documentation can be found at `www.meta-environment.org`.

2.1 The Auction Example

Consider a completely distributed auction, in which the auction master and the bidders are cooperating via a workstation in their own office. Challenges are how to synchronize bids, how to inform bidders about higher bids, and how to decide when the bidding is over. In addition, bidders may connect and disconnect from the auction whenever they want. This example is described in full detail in [2]. Since that time it has become a standard application of the ToolBus, most of all it has been used extensively in teaching and demonstrations. Its architecture is shown in Fig. 1, where ToolBus processes are represented by ellipses.

The auction is initiated by the process `Auction`, which executes the `master` tool (the user interface used by the auction master), and then handles connections and disconnections of new bidders, the introduction of new items for sale at the auction, and the actual bidding process. A delay is used to determine the end of the bidding activity per item. A `Bidder` process is created for each new `bidder` tool that connects to the auction; it describes the possible behavior of the bidder. The auxiliary process `ConnectBidder`, which handles the connection of a new bidder to the auction, consists of the following steps:

- Receive a connection request from some bidder. This may occur when someone executes a bidder tool outside the ToolBus (possibly on another computer). As part of its initialization, the bidder tool will attempt to make a connection with the ToolBus system running the auction Tscript.
- Create an instance of the process `Bidder` that defines the behavior of this particular bidder.
- Ask the bidder for its name, and send that name to the auction master.

The auxiliary process `OneSale` handles all steps needed for the sale of one item:

- Receive an event from the master tool announcing a new item for sale.
- Broadcast this event to all connected bidders, and perform one of the following four steps as long as the item is not sold:
  - Receive a new bid from one of the bidders. If the bid is too low, reject it and inform the bidder. If the bid is acceptable, inform the bidder and notify all bidders that a higher bid has been received.
  - Ask for a final bid if no bids were received during the last 10 seconds.
  - Declare the item sold if no new bids arrive within 10 seconds after asking for a final bid.
  - Connect a new bidder.

The Tscript of this auction system takes care of the issues mentioned earlier, i.e., synchronizing bids, informing bidders, and completing a sale.

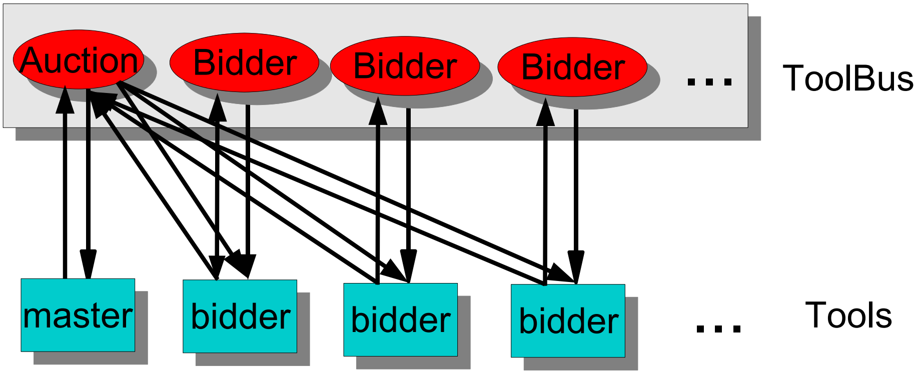


Fig. 1. Architecture of the auction application

3 mCRL2 and CADP

An mCRL2 [5] specification is built from the standard process algebraic constructs: atomic actions (including the deadlock δ and the internal action τ), alternative composition $+$, sequential composition $\cdot$ and parallel composition $\|$. One can define synchronous communication of actions. The following two operators combine data with processes. The sum operator $\sum_{d:D} p(d)$ describes the process that can execute the process $p(d)$ for values d of sort D. The conditional operator $\_ \rightarrow \_ \diamond \_$ describes the *if-then-else*. The process $b \rightarrow x \diamond y$ (where b is a boolean) has the behavior of x if b is true and the behavior of y otherwise.

Data elements are terms of some sort. Next to equational abstract data types, mCRL2 also supports built-in functional data types. Atomic actions are parametrized with data parameters, and can be provided with an absolute time stamp. A process definition is of the form $\mathsf{Pname}(x_1, \ldots, x_n) = P$, with P an mCRL2 process and $x_1, \ldots, x_n$ a list of parameters.

The mCRL2 toolset (`www.mcrl2.org`) supports formal reasoning about systems specified in mCRL2. It is based on term rewriting techniques and on formal transformation of process algebraic and data terms. mCRL2 specifications are first transformed to a linear form [5, Section 5], in a *condition-action-effect* style. The resulting specification can be simulated interactively or automatically, there are a number of symbolic optimization tools, and the corresponding Labeled Transition System (LTS) can be generated. This LTS can, in turn, be minimized modulo a range of behavioral semantics, and model checked with the mCRL2 toolset or the CADP toolset [7].

4 From Tscript to mCRL2

Both Tscript and mCRL2 are based on the process algebra ACP [3]. In spite of this common origin, the languages have some important differences. In [4], we proposed how these differences can be bridged. For instance, the binary star operation in TSCRIPT can be encoded by means of recursive equations in mCRL2. And dynamic process creation in TSCRIPT can be modeled in mCRL2 by statically fixing the maximal number of process instances that can be active simultaneously; these process instances are present from the start, and the master process divides connecting tools over these processes. And the notion of discrete time in TSCRIPT can be modeled using a tick action synchronization (cf. [27,28,29]). Here we go over two of the main differences, and show how they relate to the auction example.

Asynchronous Communication. According to the semantics of the TOOLBUS, each process created by TSCRIPT has a queue for incoming notes. A `rec-note` will inspect the note queue of the current process, and if the queue contains a note of a given form, it will remove the note and assign values to variables appearing in its argument list; these can be used later on in the process expression in which the `rec-note` occurs.

mCRL2 contains no built-in primitives for asynchronous communication. Therefore, in mCRL2, note queues are handled by a separate AsyncComm process. It also takes care of subscriptions/unsubscriptions and lets any process send any note at any time. Any process can inspect its queue for incoming notes by synchronously communicating with AsyncComm.

$$
\begin{aligned}
&\mathsf{AsyncComm}(\mathit{subscribers}{:}\mathbf{L}(\mathit{Pid}), \mathit{queues}{:}\mathbf{L}(\mathbf{L}(\mathit{Msg}))) = \\
&\quad\ \textstyle\sum_{p:\mathit{Pid}} \mathsf{r\_subscribe}(p) \cdot \mathsf{AsyncComm}(\mathit{subscribers} \lhd p, \mathit{queues}) \\
&\quad + \textstyle\sum_{p:\mathit{Pid}} \mathsf{r\_unsubscribe}(p)\cdot \\
&\qquad\quad \mathsf{AsyncComm}(\mathit{rem\_list\_elem}(\mathit{subscribers}, p), \mathit{set\_elem}(\mathit{queues}, p, [\,])) \\
&\quad + \textstyle\sum_{\mathit{note}:\mathit{Msg}} \mathsf{r\_snd\_note}(\mathit{note})\cdot \\
&\qquad\qquad \mathsf{AsyncComm}(\mathit{subscribers}, \mathit{distr\_note}(\mathit{queues}, \mathit{subscribers}, \mathit{note})) \\
&\quad + \textstyle\sum_{p:\mathit{Pid}} \sum_{\mathit{ntype}:\mathit{NoteType}} \\
&\qquad (p < \#\mathit{queues} \wedge \mathit{has\_note\_of\_type}(\mathit{queues}.p, \mathit{ntype})) \rightarrow \\
&\qquad\qquad \mathsf{s\_rec\_note}(p, \mathit{get\_first\_of\_type}(\mathit{queues}.p, \mathit{ntype}))\cdot \\
&\qquad\qquad \mathsf{AsyncComm}(\mathit{subscribers}, \mathit{set\_elem}(\mathit{queues}, p, \\
&\qquad\qquad\qquad\qquad\qquad \mathit{rem\_first\_of\_type}(\mathit{queues}.p, \mathit{ntype})))
\end{aligned}
$$

The process AsyncComm is parametrized by the list of *subscribers* (for the sake of simplicity we assume that processes can subscribe/unsubscribe to all notes simultaneously), and by the list of note queues containing the pending notes for the subscribed processes. The four summands of the process definition reflect the four actions that AsyncComm react upon.

The first two summands handle the subscription and unsubscription. A process willing to subscribe performs $\mathsf{s\_subscribe}(\mathit{id})$ action that synchronizes to the $\mathsf{r\_subscribe}(p)$ action of the first summand. This can only happen for the value of p that is equal to id, and as a result of this action the id is added to *subscribers*. In a similar way an id of the unsubscribing process is removed from *subscribers* and its queue is emptied.

The third summand says that a sent note is distributed into the queues of all subscribers. The fourth summand deals with reception of a note by a process p. It can only happen if its queue has a note of the appropriate type. In this case the first note of this type is taken from the queue, and is delivered to process p.

4.1 Structure of the Translator

The actual translation program is implemented as a sequence of transformation steps. The first step performs unfoldings of TScripts and some other syntactic sugar removals as a TScript to TScript transformation implemented in ASF [26].

The simplified TScript is then compiled by a part of the ToolBus system to an internal representation, containing a finite automata representation for each process in the TScript. Each state n of such a representation is translated to an mCRL2 process of the form

$$\begin{aligned}\mathsf{P}_n(\overrightarrow{v{:}V}) = &\textstyle\sum_{\overrightarrow{v_1{:}V_1}} c_1(\overrightarrow{v}) \to \mathsf{a}_1(\overrightarrow{f_1}(\overrightarrow{v}), \overrightarrow{v_1}) \cdot \mathsf{P}_{n(1)}(\overrightarrow{g_1}(\overrightarrow{v}, \overrightarrow{v_1}))\\ &+\\ &\ldots\\ &+ \textstyle\sum_{\overrightarrow{v_k{:}V_k}} c_k(\overrightarrow{v}) \to \mathsf{a}_k(\overrightarrow{f_k}(\overrightarrow{v}), \overrightarrow{v_k}) \cdot \mathsf{P}_{n(k)}(\overrightarrow{g_k}(\overrightarrow{v}, \overrightarrow{v_k}))\end{aligned}$$

where k is the number of outgoing transitions from state n and for any transition i such that $1 \leq i \leq k$, $n(i)$ is the next state of process P. The vector $\overrightarrow{v{:}V}$ represents the local variables of process P in state n. The vectors $\overrightarrow{v_i{:}V_i}$ represent the input variables (if any) that are being assigned by performing the action a_i of transition i. These variables are used to determine the values of the local variables in the next state using the vector of functions $\overrightarrow{g_i}(\overrightarrow{v}, \overrightarrow{v_i})$.

As the final step, the standard parts, like the AsyncComm process, are added to the generated mCRL2 model. As a result the generated mCRL2 model performs the actions of the TScript. By performing deadlock or reachability analysis one can obtain a trace to an undesirable state of the mCRL2 model. The actions in this trace map directly to the actions of the TScript.This gives a possibility to locate and correct the problem in the TScript. The modified TScript can be translated to mCRL2 again and the analysis can be repeated. In this iterative way one can get a TScript where all formulated behavioral properties are satisfied. Executing this TScript and performing some tests of the working system can reveal additional problems that can also be formulated as behavioral properties and checked with the mCRL2 level.

5 Analysis of the Auction System

We translated the TScript of the auction system to mCRL2 using the prototype translator. A small example is given in Figure 2.

The structure of the resulting mCRL2 model is presented in Figure 3. Here each ToolBus process is represented by an mCRL2 process, and an extra process AsyncComm is added to model the asynchronous communication of ToolBus.

`Tscript` fragment

```
    tool bidder is {}
 %% Declaration of the tool type "bidder"

    process ConnectBidder is
       let Bidder : bidder in
          rec-connect(Bidder?) . snd-msg(new(Bidder)) ...
       endlet
 %% Suppose the auction runs on location ($HOST, $PORT). Then a tool of
 %% type "bidder" can be launched on any client by entering the command:
 %% wish-adapter $HOST $PORT -TB_TOOL_NAME bidder ...
```

becomes `mCRL2` fragment

```
    sort bidder; Msg = struct new(bidder)|any-higher-bid|... ;
 %% Tool types are represented in mCRL2 as sorts. "Msg" is a sort
 %% added to the mCRL2 specification to represent a Tscript message.

    act  rec-connect : bidder; snd-msg : Msg;
 %% Some mCRL2 declarations of Tscript built-in actions.

    proc ConnectBidder() = sum(Bidder:bidder,
       rec-connect(Bidder) . ConnectBidder1(Bidder));
 %% "sum(Bidder:bidder" introduces a local variable "Bidder" of sort "bidder".
 %% "rec-connect" communicates with "snd-connect" defined in an environment.

    ConnectBidder1(Bidder : bidder) = snd-msg(new(Bidder)) ...
 %% Here "Bidder" is initialised with the value received by "rec-connect".
```

Fig. 2. Fragment of a translation from TSCRIPT into mCRL2

The mCRL2 translation of the auction TSCRIPT has been analyzed[1] for the presence of deadlocks and some other behavioral properties. This revealed two deadlocks and a race condition. Moreover, we encountered a possible Denial of Service attack. We proposed fixes for the detected problems and verified that with these fixes the system behaves correctly.

Finding 1: two deadlocks due to a fast disconnect. One deadlock occurs when a newly connected bidder disconnects immediatly instead of sending its name. In this case the newly created `Bidder` process handles the disconnect, and the `Master` process keeps waiting for the bidder's name forever. This problem can be resolved by postponing the creation of the `Bidder` process till after the reception of the name of the new bidder.

Another problem occurs when the `Master` process is busy with `OneSale`, and a new bidder connects to the system. After executing `ConnectBidder(Mid,Bid?)`, the `Master` process attempts to do `snd-msg(Bid,new-item(Descr,HighestBid))`. This has to synchronize with the `rec-msg` of the connected new bidder. In case that bidder has already disconnected by that time, the synchronization is impossible and the `Master` process and the whole system deadlocks. Many solutions are possible for this problem. The one we chose consists of two parts.

[1] The source code of the auction script and the full mCRL2 model can be found at `www.win.tue.nl/~yusenko/sources/Auction/sources.zip`

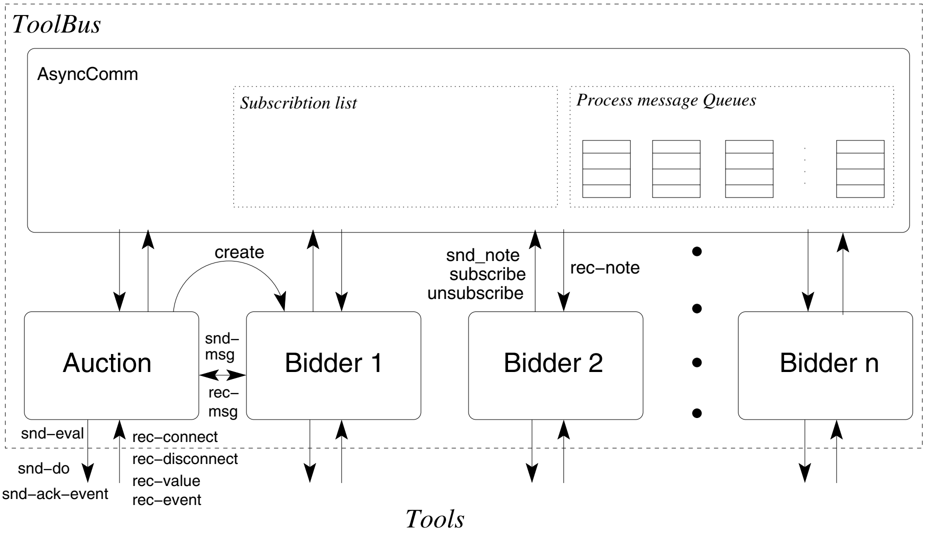


Fig. 3. Auction TScript in mCRL2

1. The `Bidder` process has to perform a `rec-msg` before disconnecting. This patch alone, however, does not solve the problem, since it introduces another one. In case the bidder tool connects *not* during an ongoing sale, the `rec-msg` would wait for synchronization forever. That is why we need another patch as well.
2. When a new bidder connects at a moment that there is no sale, the `Master` process does `snd-msg(Bid,no-new-item)`. The newly-created `Bidder` process waits for either `new-item` or a `no-new-item` message before proceeding further, or receiving a disconnect from its tool.

After bringing this fix into the TScript, we could regenerate the mCRL2 model and verify that this deadlock has been resolved.

Finding 2: a missing ***`unsubscribe`****.* The `Bidder` process contains no `unsubscribe` commands, also not before successful termination. This situation can be seen as a violation of the stylistic constraint that every subscribe command has a corresponding unsubscribe command. We found this situation by generating the underlying LTS of our mCRL2 model with the `lps2lts` tool. The tool reported a deadlock situation: due to the way we modeled the process creation/termination mechanism in mCRL2, the missing `unsubscribe` command lead to a deadlock. To be more precise, in case a new `Bidder` tool connects to the system after the missing `unsubscribe` command, the corresponding `Bidder` process cannot perform `subscribe` any longer. We resolved this problem by adding the missing `unsubscribe`.

Finding 3: buying the next item while bidding for the previous one. A third and more serious error that we detected is that a bidder can, for a very short time

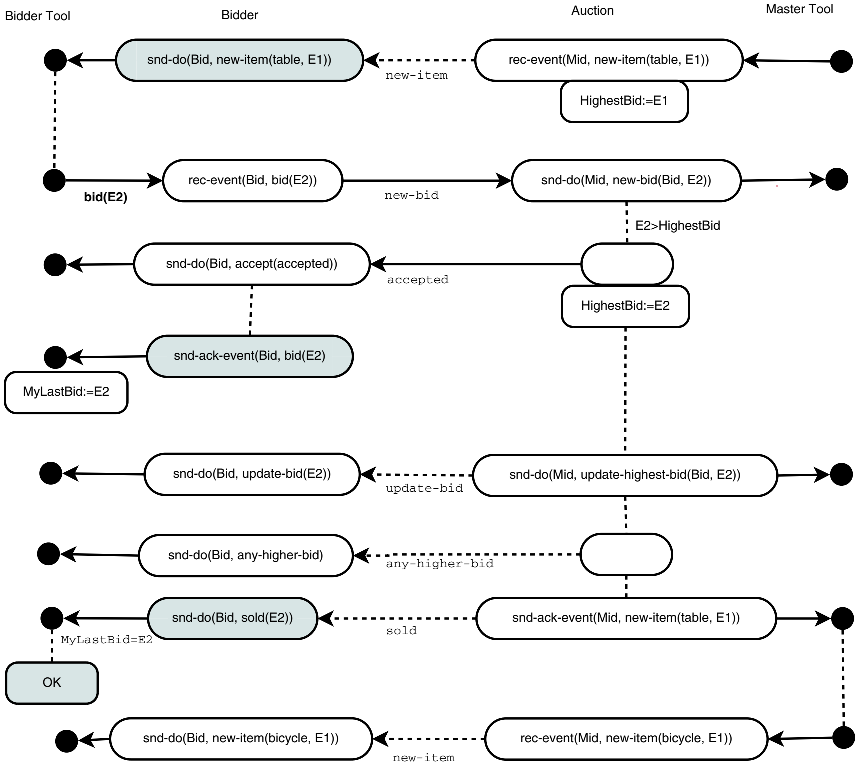

Bidder Tool
Bidder
Auction
Master Tool
snd-do(Bid, new-item(table, E1))
new-item
rec-event(Mid, new-item(table, E1))
HighestBid:=E1
bid(E2)
rec-event(Bid, bid(E2))
new-bid
snd-do(Mid, new-bid(Bid, E2))
E2>HighestBid
snd-do(Bid, accept(accepted))
accepted
HighestBid:=E2
snd-ack-event(Bid, bid(E2)
MyLastBid:=E2
snd-do(Bid, update-bid(E2))
update-bid
snd-do(Mid, update-highest-bid(Bid, E2))
snd-do(Bid, any-higher-bid)
any-higher-bid
snd-do(Bid, sold(E2))
sold
snd-ack-event(Mid, new-item(table, E1))
MyLastBid=E2
OK
snd-do(Bid, new-item(bicycle, E1))
new-item
rec-event(Mid, new-item(bicycle, E1))

Fig. 4. Normal external behavior

slot, bid for the last item that has already been sold, while the master interprets this as a bid for the next item.

We demonstrate this race condition with a small example. Suppose that the auction master sells a table and a bicycle, both for the price of E1. Bidder B1 intends to bid E2 (a larger amount than E1) for the table. In the next paragraphs we explain the desired behavior of the auction, and possible erroneous behavior that may occur in this situation.

The desired scenario for the aforementioned example is as follows. After a new item (the table) is presented at the auction for the price of E1, the bidder bids E2. The bid is accepted, and the bidder is informed about this fact. Then every bidder receives a note that the current price is raised to E2. Bidder B1 does not bid anymore, receives a note any-higher-bid, and receives a note that the item is sold for the price of E2. From this bidder B1 can derive that he has bought the table. This sequence of events is depicted in Fig. 4.

However, using model checking, we found the following erroneous scenario. After the table is presented at the auction for the price of E1, the item is sold for this price. During the selling of that item, three broadcasts are performed

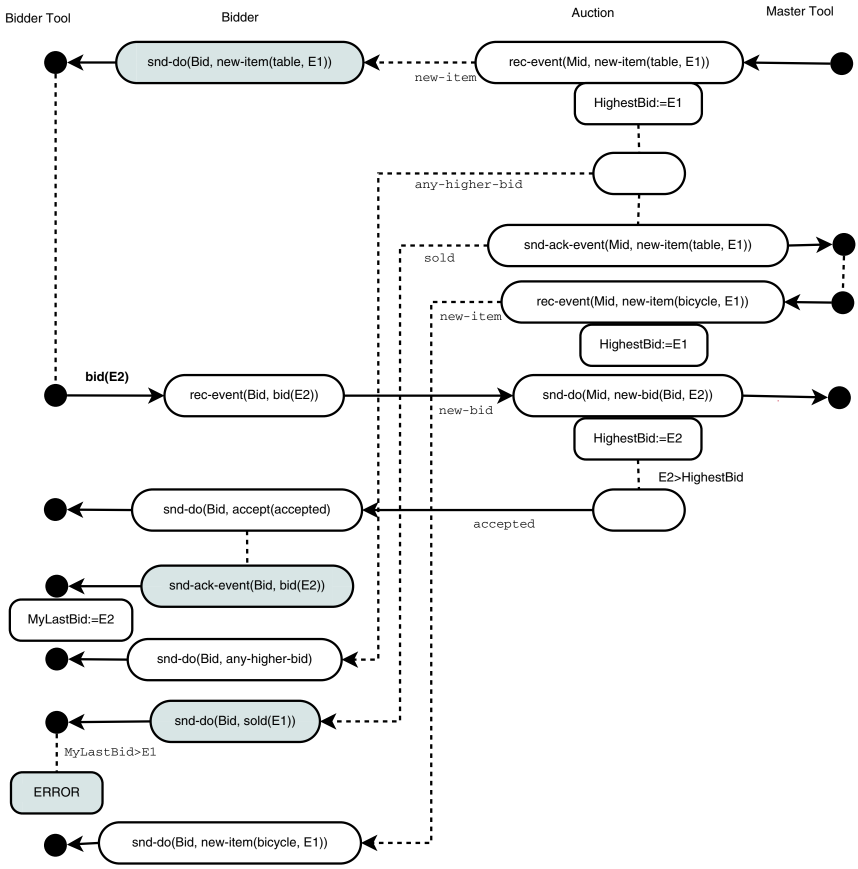

Bidder Tool
Bidder
Auction
Master Tool
snd-do(Bid, new-item(table, E1))
new-item
rec-event(Mid, new-item(table, E1))
HighestBid:=E1
any-higher-bid
snd-ack-event(Mid, new-item(table, E1))
sold
rec-event(Mid, new-item(bicycle, E1))
new-item
HighestBid:=E1
bid(E2)
rec-event(Bid, bid(E2))
new-bid
snd-do(Mid, new-bid(Bid, E2))
HighestBid:=E2
E2>HighestBid
snd-do(Bid, accept(accepted)
accepted
snd-ack-event(Bid, bid(E2))
MyLastBid:=E2
snd-do(Bid, any-higher-bid)
snd-do(Bid, sold(E1))
MyLastBid>E1
ERROR
snd-do(Bid, new-item(bicycle, E1))

Fig. 5. Erroneous external behavior

to the bidders: `new-item`, `any-higher-bid`, and `sold`. Bidder `B1` bids `E2` under the illusion that he bids on the table, because the note `sold` has not arrived yet. The bid gets accepted, but the auction master thinks that this is a bid for the next item, being the bicycle. This sequence of events is depicted in Fig. 5.

To find this issue, we used the following property:

> If a `Bidder` tool bids m and this bid gets accepted (`accept(accepted)` message is received), then the following `sold(`n`)` message it receives should be such that $m \leq n$.

We used two methods to check this property. One way is in formulating the property as a μ-calculus [30] formula and performing on-the-fly model checking with the mCRL2 toolset or with the CADP tool `evaluator` [7]. Another way is in adding an observer process in parallel to the system. The process will observe the relevant actions in the way the environment (`Bidder` tool) does it. In case the property violation is detected, the observer performs the `s_error` action. The presence of this action and a shortest trace to it can be detected with the mCRL2 tool `lps2lts` during the LTS generation. Concretely, the relevant part of the observer process look as follows:

$$
\begin{array}{l}
\mathsf{ToolBidder\_1}(\mathit{tid}{:}\mathit{TId}, \mathit{accepted}{:}\mathit{Bool}, \mathit{my\_last\_bid}{:}\mathit{Int}) = \\
\quad \sum_{\mathit{msg}:\mathit{Msg}} \mathsf{r\_ack\_event}(\mathit{tid}, \mathit{msg})\cdot \\
\qquad\qquad (\mathit{accepted} \wedge \mathit{is\_bid1}(\mathit{msg})) \rightarrow \mathsf{ToolBidder\_1}(\mathit{tid}, \mathit{false}, \mathit{bid1\_0}(\mathit{msg})) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \diamond\, \mathsf{ToolBidder\_1}(\mathit{tid}, \mathit{accepted}, \mathit{my\_last\_bid}) \\
+ \sum_{\mathit{msg}:\mathit{Msg}} \mathsf{r\_do}(\mathit{tid}, \mathit{msg})\cdot \\
\qquad\qquad (\mathit{is\_sold0}(\mathit{msg}) \rightarrow \\
\qquad\qquad\quad ((\mathit{my\_last\_bid} > \mathit{sold0\_0}(\mathit{msg})) \rightarrow \mathsf{s\_error}\cdot \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{ToolBidder\_1}(\mathit{tid}, \mathit{accepted}, 0) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \diamond\, \mathsf{ToolBidder\_1}(\mathit{tid}, \mathit{accepted}, 0) \\
\qquad\qquad\quad) \\
\qquad\qquad \diamond\, \mathit{is\_accept0}(\mathit{msg}) \rightarrow \\
\qquad\qquad\quad ((\mathit{accept0\_0}(\mathit{msg}) = \mathit{trm\_accepted}) \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{ToolBidder\_1}(\mathit{tid}, \mathit{true}, \mathit{my\_last\_bid}) \\
\qquad\qquad\qquad\qquad\qquad \diamond\, \mathsf{ToolBidder\_1}(\mathit{tid}, \mathit{false}, \mathit{my\_last\_bid}) \\
\qquad\qquad\quad) \\
\qquad\qquad \diamond\, \mathsf{ToolBidder\_1}(\mathit{tid}, \mathit{accepted}, \mathit{my\_last\_bid}) \\
\qquad\qquad) \\
+ \ldots
\end{array}
$$

Here the first summand reacts to the r_ack_event action and, in case the last bid has been accepted, sets the value of the *my_last_bid* parameter to the amount in the last bid. The second summand reacts to r_do. Here two messages of the TSCRIPT are important: `sold` and `accept`. The latter one is used to set the *accepted* parameter to represent the fact that our last bid has been accepted. The former one leads to the check if our property $\mathit{my\_last\_bid} > \mathit{sold0\_0}(\mathit{msg})$ holds. In case it does not, the s_error action is executed.

We believe that this problem can be resolved in many different ways. However, not much can be done without increasing the level of detail in the communication protocol between the auction TSCRIPT and the tools. We propose a solution to this problem without changing this protocol. We extend the `bid` message that is synchronously communicated from the `Bidder` process to the `Auction` process with the description of the item for which the bid is valid. In case the `Auction` process receives a bid for a wrong item, it rejects this bid.

Additionally we had to add an extra condition to the case when a new bidder connects during an ongoing sale. Namely, we make it impossible to connect in this way if the item has already been sold. In this case the connection is performed after this sale round is finished.

To verify the fact that this solution actually works we had to decorate the `accept` and `sold` messages that are sent to the `Bidder` tool with the description and the amount information. An important assumption for our solution to work is that the consecutive sale items must have different descriptions. We could verify that the resulting TSCRIPT does not have the erroneous behavior.

An important observation related to this problem has been proposed by an anonymous reviewer. The root of the problem lays in the fact that a bid comes into a race condition with the `sold` message. In case a bid comes late and there is no next item to be sold, the bidder will have to wait for the rejection of its bid forever. To avoid this problem the reviewer proposed to allow a choice between `snd-msg(bid(...,...))` and `rec-note(sold(...))`. We implemented this fix and checked that the rejection is always received by the bidder tool.

Finding 4: infinite queues. Although using on-the-fly model checking we could detect some important issues, we could not analyze the entire behavior of the auction TScript, due to the fact that its LTS is infinite. We could handle some sources of infinity, like infinite domains for data types, by bounding these domains.

Another source of infinity has to do with the asynchronous communication and queues for notes. For example, one process may keep sending notes while another process is not willing to receive them. Such a situation can happen in case one of the bidders keeps bidding very actively. The following part of the `Bidder` process illustrates such a phenomenon.

```
( rec-event(Bid, bid(Amount?)).
  ...
  + rec-note(update-bid(Amount?)) .
    snd-do(Bid, update-bid(Amount))
  + rec-note(any-higher-bid) . snd-do(Bid, any-higher-bid)
  + rec-disconnect(Bid).delta ) *
```

In case a `Bidder` tool is active in sending bids, the first alternative is enabled and can be chosen. As a result, the parallel OneSale process adds an `update-bid` note to the message queue of our `Bidder` process. For any fixed-size note queue, an overflow can be reached by executing a sufficiently large number of first alternatives without ever taking the second one.

This can be seen as a problem that has to be solved by the scheduler of the ToolBus. Another way to look at this problem is to see it as a possibility of a Denial of Service attack, leading to an overflow.

Some of such queue size problems can be tackled with the help of timing. We could impose that no time can progress as long as a process can receive a note (so-called maximal progress). However, the aforementioned problem can happen without reception of any notes and, therefore, without any progress of time at all. To solve this issue we chose to limit the number of bids per time unit that the `Bidder` process is willing to accept from its tool. By combining the two timing restrictions into the mCRL2 model, we could get to a finite LTS.

6 Conclusions and Future Work

Our general aim is to have a process algebra-based software development environment where both formal verification and production of an executable system is possible. We implemented a prototype translation from TScript to mCRL2.

This translation makes it possible to verify TSCRIPT in an automated fashion, and to explore behavioral properties of executable software systems that have been built with the TOOLBUS.

We automatically translated a standard TSCRIPT application, a distributed auction, to mCRL2, and analyzed it using on-the-fly and symbolic model checking techniques. As a result, four flaws were detected in the original TSCRIPT description of this auction system. We could fix all the issues, and verified the correctness of the fixed TSCRIPT by automatically translating it to mCRL2. We could also execute and test the fixed model to ensure it still works.

In the future we aim at applying the presented techniques to analyze a large existing TSCRIPT with the help of model checking. An example of such a system is the the ASF+SDF Meta-Environment [25,26]. It is also of our interest to develop a new TSCRIPT from scratch in a way that formal verification with mCRL2 contributes to every stage of the development process.

References

1. Bergstra, J.A., Klint, P.: The ToolBus coordination architecture. In: Hankin, C., Ciancarini, P. (eds.) COORDINATION 1996. LNCS, vol. 1061, pp. 75–88. Springer, Heidelberg (1996)
2. Bergstra, J.A., Klint, P.: The discrete time ToolBus - a software coordination architecture. Sci. Comput. Program. 31(2-3), 205–229 (1998)
3. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control 60(1-3), 109–137 (1984)
4. Fokkink, W., Klint, P., Lisser, B., Usenko, Y.S.: Towards formal verification of ToolBus scripts. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 160–166. Springer, Heidelberg (2008)
5. Groote, J.F., Mathijssen, A.H.J., Reniers, M.A., Usenko, Y.S., van Weerdenburg, M.J.: The formal specification language mCRL2. In: Proc. Methods for Modelling Software Systems. Dagstuhl Seminar Proceedings, vol. 06351 (2007)
6. Bergstra, J.A., Heering, J., Klint, P.: Module algebra. J. ACM 37(2), 335–372 (1990)
7. Garavel, H., Mateescu, R., Lang, F., Serwe, W.: CADP 2006: A toolbox for the construction and analysis of distributed processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
8. Groote, J.F., Willemse, T.A.C.: Parameterised boolean equation systems. Theor. Comput. Sci. 343(3), 332–369 (2005)
9. Wing, J.M.: Writing Larch interface language specifications. ACM Trans. Program. Lang. Syst. 9(1), 1–24 (1987)
10. Guaspari, D., Marceau, C., Polak, W.: Formal verification of Ada programs. IEEE Trans. Software Eng. 16(9), 1058–1075 (1990)
11. Zhao, J., Rinard, M.C.: Pipa: A behavioral interface specification language for aspectJ. In: Pezzé, M. (ed.) FASE 2003. LNCS, vol. 2621, pp. 150–165. Springer, Heidelberg (2003)
12. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)

13. Larsson, D., Alexandersson, R.: Formal verification of fault tolerance aspects. In: Supplementary Proceedings of International Symposium on Software Reliability Engineering (ISSRE) Conference, pp. 279–280. IEEE, Los Alamitos (2005)
14. Hilderink, G.H., Bakkers, A.W.P., Broenink, J.F.: A distributed real-time Java system based on CSP. In: Proc. ISORC 2000, pp. 400–410. IEEE, Los Alamitos (2000)
15. Orlic, B., Broenink, J.F.: Design Principles of the SystemCSP Software Framework. In: McEwan, A.A., Ifill, W., Welch, P.H. (eds.) CPA 2007, pp. 207–228 (2007)
16. Hopcroft, P.J., Broadfoot, G.H.: Combining the box structure development method and CSP for software development. Electr. Notes Theor. Comput. Sci. 128(6), 127–144 (2005)
17. Prowell, S.J., Poore, J.H.: Foundations of sequence-based software specification. IEEE Trans. Software Eng. 29(5), 417–429 (2003)
18. Broadfoot, G.H.: Asd case notes: Costs and benefits of applying formal methods to industrial control software. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 548–551. Springer, Heidelberg (2005)
19. Doxsee, S., Gardner, W.B.: Synthesis of C++ software from verifiable CSPm specifications. In: Proc. ECBS 2005, pp. 193–201 (2005)
20. Beek, B., Man, K.L., Reniers, M., Rooda, K., Schiffelers, R.: Syntax and consistent equation semantics of hybrid χ. J. Log. Algebr. Program. 68(1-2), 129–210 (2006)
21. Braspenning, N.C.W.M., van de Mortel-Fronczak, J.M., Rooda, J.E.: A model-based integration and testing method to reduce system development effort. Electr. Notes Theor. Comput. Sci. 164(4), 13–28 (2006)
22. Diertens, B.: Simulation and animation of process algebra specifications. Technical Report P9713, University of Amsterdam (1997)
23. Diertens, B.: Software (re-)engineering with PSF III: An IDE for PSF. Technical Report PRG0708, University of Amsterdam (2007)
24. van der Brand, M., de Jong, H., Klint, P., Olivier, P.: Efficient annotated terms. Softw., Pract. Exper. 30(3), 259–291 (2000)
25. Klint, P.: A meta-environment for generating programming environments. ACM TOSEM 2(2), 176–201 (1993)
26. den van Brand, M.G.J., van Deursen, A., Heering, J., de Jong, H.A., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The ASF+SDF meta-environment: A component-based language development environment. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 365–370. Springer, Heidelberg (2001)
27. Fokkink, W., Ioustinova, N., Kesseler, E., van de Pol, J., Usenko, Y.S., Yushtein, Y.A.: Refinement and verification applied to an in-flight data acquisition unit. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 1–23. Springer, Heidelberg (2002)
28. Blom, S., Ioustinova, N., Sidorova, N.: Timed verification with μCRL. In: Broy, M., Zamulin, A.V. (eds.) PSI 2003. LNCS, vol. 2890, pp. 178–192. Springer, Heidelberg (2004)
29. Wijs, A.: Achieving discrete relative timing with untimed process algebra. In: ICECCS, pp. 35–46. IEEE, Los Alamitos (2007)
30. Mateescu, R., Sighireanu, M.: Efficient on-the-fly model-checking for regular alternation-free mu-calculus. Sci. Comput. Program. 46(3), 255–281 (2003)

Executable Interface Specifications for Testing Asynchronous Creol Components*

Immo Grabe[1], Marcel Kyas[2,**], Martin Steffen[3], and Arild B. Torjusen[3]

[1] Christian-Albrechts University Kiel, Germany
[2] Department of Computer Science, Freie Universität Berlin, Germany
[3] Department of Informatics, University of Oslo, Norway
igb@informatik.uni-kiel.de, marcel.kyas@fu-berlin.de,
{msteffen,aribraat}@ifi.uio.no

Abstract. We propose and explore a formal approach for black-box testing asynchronously communicating components in open environments. Asynchronicity poses a challenge for validating and testing components. We use Creol, a high-level, object-oriented language for distributed systems and present an interface specification language to specify components in terms of traces of observable behavior.

The language enables a concise description of a component's behavior, it is executable in rewriting logic and we use it to give test specifications for Creol components. In a specification, a clean separation between interaction under control of the component or coming from the environment is central, which leads to an assumption-commitment style description of a component's behavior. The assumptions schedule the inputs, whereas the outputs as commitments are tested for conformance with the specification. The asynchronous nature of communication in Creol is respected by testing only up-to a notion of observability. The existing Creol interpreter is combined with our implementation of the specification language to obtain a specification-driven interpreter for testing.

1 Introduction

To reason about open distributed systems and predicting their behavior is intrinsically difficult. A reason for that is the inherent asynchronicity and the resulting non-determinism. It is generally accepted that the only way to approach complex systems is to "divide-and-conquer", i.e., consider components interacting with their environment. Abstracting from internal executions, their black-box behavior is given by interactions at their *interface.* In this paper we use Creol [1], a programming and modeling language for distributed systems based on concurrent, active objects communicating via asynchronous method calls.

* Part of this work has been supported by the EU-project IST-33826 *Credo: Modeling and analysis of evolutionary structures for distributed services* and the German-Norwegian DAAD-NWO exchange project *Avabi* (Automated validation for behavioral interfaces of asynchronous active objects).

** M. Kyas' research was partly performed while employed at University of Oslo.

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 324–339, 2010.

To describe and test Creol components, we introduce a concise specification language over communication labels. The expected behavior is given as a set of traces at the interface. Both input and output interactions are specified but play quite different roles. As input events are not under the control of the object, but of the environment, input is considered as assumptions about the environment whereas output describes commitments of the object. For input interactions, we ensure that the specified assumptions on the environment are fulfilled by *scheduling* the incoming calls in the order specified, while for output events, which are controlled by the component, we *test* that the events occur as specified. An expression in the specification language thus gives an assumption-commitment style specification[2] for a component by defining the valid observable output behavior under the assumption of a certain scheduling of the input. Scheduling and testing of a component is done by synchronizing the execution of the component with the specification. As a result, the scheduling is enforced in the execution of the component and the actual outgoing interactions from the component are tested against the output labels in the specification. This gives a framework for testing whether an implementation of a component conforms with the interface specification. Incorrect or nonconforming behavior of the component under a given scheduling is reported as an error.

It is important in the specification, to carefully distinguish between the interactions which are scheduled and those for which the component is responsible and which are checked for conformance. We do so by formalizing *well-formedness* conditions on specifications. Well-formedness enforces a syntactic distinction between input and output specifications and, in addition, assures that only "meaningful" traces, i.e., those corresponding to possible behavior, can be specified. Besides that, the specification language captures two crucial features of the interface behavior of Creol objects. First, Creol allows to dynamically create objects and threads (via asynchronous method calls), which gives rise to dynamic scoping. This is reflected in the interface behavior by scope extrusion and the specification language allows to express *freshness* of communicated object and thread references. Second, due to the asynchronous nature of the communication model, the order in which outgoing messages from a component are observed by an external observer does not necessarily reflect the order in which they where actually sent. We take this asynchronous message passing into account by only considering trace specifications up-to an appropriate notion of *observational equivalence.*

Contributions. The paper contains the following contributions: We *formalize* the interface behavior of a concurrent, object-oriented, language plus a corresponding behavioral interface specification language in Sect. 2 and Sect. 3. This gives the basis for testing active Creol objects, where a test environment can be simulated by execution of the specifications. Sect. 4 explains how to compose a Creol program and a specification and how to use this for testing. Furthermore, the existing Creol interpreter is extended with the *implementation* of the specification language. This yields a specification-driven interpreter for testing asynchronous Creol components. The implementation is described in Sect. 5

2 The Creol Language

Creol [3,1] is a high-level object-oriented language for distributed systems, featuring active objects and asynchronous method calls. Concentrating on the core features, we elide inheritance, dynamic class upgrades, etc. They would complicate the interface description, but not alter the basic ideas presented here.

The Creol-language features active objects and its communication model is based on exchanging messages *asynchronously.* This is in contrast with object-oriented languages based on multi-threading, such as *Java* or $C^{\#}$, which use "synchronous" message passing in which the calling thread inside one object blocks and control is transferred to the callee. Exchanging messages asynchronously decouples caller and callee, which makes that mode of communication advantageous in a distributed setting. On the receiver side, i.e., at the side of the callee, each object possesses an input "queue" in which incoming messages are waiting to be served by the object. To avoid uncontrolled interference, each object acts as a *monitor*, i.e., at most one method body is executing at each point in time. The choice, which method call in the input queue is allowed to enter the object next is *non-deterministic.*

After presenting the abstract syntax in the next section, we sketch the operational semantics, concentrating on the external behavior, i.e., the message exchange with the environment.

2.1 Syntax

The abstract syntax of the calculus, which is in the style of standard object calculi [4], is given in Tab. 1. It distinguishes between *user* syntax and *run-time* syntax, the latter underlined. The user syntax contains the phrases in which programs are written; the run-time syntax contains syntactic material additionally needed to express the behavior of the executing program in the operational semantics.

The basic syntactic category of names n, which count among the values v, represents references to classes, to objects, and to threads. To facilitate reading, we allow ourselves to write o and its syntactic variants for names referring to objects, c for classes, and n when being unspecific. Technically, the disambiguation between the different roles of the names is done by the type system and the abstract syntax of Tab. 1 uses the non-specific n for names. The unit value is represented by $()$ and x stands for variables, i.e., local variables and formal parameters, but not instance variables.

A *component* C is a collection of classes, objects, and (named) threads, with $\mathbf{0}$ representing the empty component. The sub-entities of a component are composed using the parallel-construct $\parallel$. The entities executing in parallel are the named threads $n\langle t\rangle$, where t is the code being executed and n the name of the thread. The name n of the thread is, at the same time, the future reference under which the result value of t, if any, will be available. In this paper, when describing the interface behavior, we restrict ourselves to the situation where the component consists of one object only, plus arbitrary many threads/method

Table 1. Abstract syntax

$C ::= \mathbf{0} \mid C \parallel C \mid \underline{\nu(n{:}T).}C \mid n\llbracket O \rrbracket \mid \underline{n[n, F, L]} \mid \underline{n\langle t \rangle}$	component
$O ::= F, M$	object
$M ::= l = m, \dots, l = m$	method suite
$F ::= l = f, \dots, l = f$	fields
$m ::= \varsigma(n{:}T).\lambda(x{:}T, \dots, x{:}T).t$	method
$f ::= \varsigma(n{:}T).\lambda().v \mid \varsigma(n{:}T).\lambda().\bot_{n'}$	field
$t ::= v \mid \mathsf{stop} \mid \mathsf{let}\, x{:}T = e \,\mathsf{in}\, t$	thread
$e ::= t \mid \mathsf{if}\, v = v \,\mathsf{then}\, e \,\mathsf{else}\, e \mid \mathsf{if}\, \mathit{undef}(v.l()) \,\mathsf{then}\, e \,\mathsf{else}\, e$	expr.
$\mid\; v@l(\vec{v}) \mid \underline{v.l(\vec{v})} \mid v.l() \mid v.l := \varsigma(s{:}n).\lambda().v$	
$\mid\; \mathsf{new}\, n \mid \mathsf{claim}@(n, n) \mid \underline{\mathsf{get}@n} \mid \mathsf{suspend}(n) \mid \underline{\mathsf{grab}(n)} \mid \underline{\mathsf{release}(n)}$	
$v ::= x \mid n \mid ()$	values
$L ::= \bot \mid \top$	lock status

bodies under execution. A class $c\llbracket O \rrbracket$ carries a name c and defines its methods and fields in O. An object $o[c, F, L]$ with identity o keeps a reference to the class c it instantiates, stores the current value F of its fields, and maintains a *binary lock* L indicating whether any code is currently active inside the object (in which case the lock is taken) or not (in which case the lock is free). The symbols $\top$ and $\bot$ indicate that the lock is taken or free respectively. Of the three kinds of entities at the component level—threads $n\langle t \rangle$, classes $n\llbracket O \rrbracket$, and objects $o[c, F, L]$—only the threads are *active*, executing entities, being the target of the reduction rules. The objects, in contrast, store the state in their fields or instance variables, whereas the classes are constant entities specifying the methods.

The named threads $n\langle t \rangle$ are incarnations of method bodies "in execution". Each thread belongs to one specific object "inside" which it executes, i.e., whose instance variables it has access to. Object locks are used to rule out unprotected concurrent access to the object states: Though each object may have more than one method body incarnation partially evaluated, at each time point at most one of those bodies (the lock owner) can be active inside the object. Method calls in Creol are issued *asynchronously,* i.e., the calling thread continues executing and the code of the method being called is computed concurrently in a new thread located in the callee object. The ν-operator is used for hiding and dynamic scoping, as known from the π-calculus [5]. In a component $C = \nu(n{:}T).C'$, the scope of the name n (of type T) is restricted to C' and unknown outside C. ν-binders are introduced when dynamically creating new named entities, i.e., when instantiating new objects or new threads. The scope of a ν-binder is dynamic, when the name is communicated by message passing, the scope is enlarged.

Besides components, the grammar specifies the lower level syntactic constructs, in particular, methods, expressions, and (unnamed) threads, which are basically sequences of expressions. A method $\varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t$ provides the method body t abstracted over the ς-bound "self" parameter, here s, and the formal parameters $\vec{x}$. For uniformity, fields are represented as methods without parameters (except self), with a body being either a value or yet undefined. Note that the methods are stored in the classes but the fields are kept in the objects. In freshly created

Table 2. Internal steps

$o[c, F, L] \parallel n\langle \mathsf{let}\, x{:}T = o.l() \;\mathsf{in}\; t\rangle \xrightarrow{\tau} o[c, F, L] \parallel n\langle \mathsf{let}\, x{:}T = F.l(o)() \;\mathsf{in}\; t\rangle$	FLookup
$o[c, F, L] \parallel n\langle \mathsf{let}\, x{:}T = o.l := v \;\mathsf{in}\; t\rangle \xrightarrow{\tau} o[c, F.l := v, L] \parallel n\langle \mathsf{let}\, x{:}T = o \;\mathsf{in}\; t\rangle$	FUpdate
$n\langle \mathsf{let}\, x : T = o@l(\vec{v}) \;\mathsf{in}\; t\rangle \xrightarrow{\tau}$ $\nu(n'{:}T)(n\langle \mathsf{let}\, x : T = n' \;\mathsf{in}\; t\rangle \parallel n'\langle \mathsf{let}\, x : T = o.l(\vec{v}) \;\mathsf{in}\; \mathsf{stop}\rangle)$	CallO$_i$

objects, the lock is free, and all fields carry the undefined reference $\bot_c$, where class name c is the (return) type of the field.

We use f for instance variables or fields and $l = \varsigma(s{:}T).\lambda().v$, resp. $l = \varsigma(s{:}T).\lambda().\bot_c$ for field variable definition (l is the label of the field). Field access is written as $v.l()$ and field update as $v'.l := \varsigma(s{:}T).\lambda().v$. By convention, we abbreviate the latter constructs by $l = v$, $l = \bot_c$, $v.l$, and $v'.l := v$. Note that the construct $v.l()$ is used for field access only, but not for method invocation. The expression $v@l(\vec{v})$ denotes an asynchronous method call, $v.l(\vec{v})$ is run-time syntax for a synchronous call and hence not available for the user. We also use $v_\bot$ to denote either a value v or a symbol $\bot_c$ for being undefined. Direct access to fields across object boundaries is forbidden by convention, and we do not allow method update. Instantiation of a new object from class c is denoted by $\mathsf{new}\, c$.

The expression $o@l(\vec{v})$ denotes an asynchronous method call, where the caller creates a new thread/future reference and continues its execution. The further expressions claim, get, suspend, grab, and release deal with synchronization. They take care of releasing and acquiring the lock of an object appropriately. As they work pretty standard and as lock-handling is not visible at the interface (and thus does not influence the development), we omit describing them in detail here and refer to the longer version [6].

2.2 Operational Semantics

The operational semantics of a program being tested is given in two stages: steps *internal* to the program, and those occurring at the interface. The two stages correspond to the rules of Tab. 2 and 4. The internal rules deal with steps not interacting with the object's environment, such as sequential composition, conditionals, field lookup and update, etc. The rules are standard and most are omitted here. We also omit the definition of structural congruence here, specifying standard structural properties such as associativity, commutativity, and basic facts about scoping. The elided rules can be found in the long version [6]. The communication labels, the basic building blocks of the interface interactions, are given in Tab. 3. A component or object exchanges information with the environment via *call*- and *return*-labels, and the interactions is either incoming or outgoing (marked ? resp. !). The basic label $n\langle call\ o.l(\vec{v})\rangle$ represents a call of method l in object o. In that label, n is a name identifying the thread that executes the method in the callee and is therefore the (future) reference under which the result of the method call will be available (if ever) for the caller. The incoming label $n\langle \mathit{return}(v)\rangle?$ hands the value from the corresponding call back to the object, which renders it ready to be read. Its counterpart, the outgoing return, passes the value to the environment. Besides that, labels can be prefixed by bindings of the form $\nu(n{:}T)$ which express freshness of the transmitted name, i.e., scope extrusion. As usual, the order of such bindings does not play a role.

Given a basic label $\gamma = \nu(\Xi).\gamma'$ where Ξ is a name context such that $\nu(\Xi)$ abbreviates a sequence of single $n{:}T$ bindings and where γ' does not contain any binders, we call γ' the *core* of the label and refer to it by $\lfloor\gamma\rfloor$. We define the core analogously for receive and send labels. The free names $\mathit{fn}(a)$ and the bound names $\mathit{bn}(a)$ of a label a are defined as usual, whereas $\mathit{names}(a)$ refer to all names of a.

The interface behavior is given by the 4 rules of Tab. 4, which correspond to the 4 different kinds of labels, a call or a return, either incoming or outgoing. The external steps are given as transitions of the form $\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash \acute{C}$, where Ξ and $\acute{\Xi}$ represents the assumption/commitment contexts of C before and after the step, respectively. In particular, the context contains the identities of the objects and threads known so far, and the corresponding typing information. This information is *checked* in incoming communication steps, and updated when performing a step (input or output).

These two operations are captured by the following notation

$$\Xi \vdash a : T \qquad \text{and} \qquad \Xi + a \tag{1}$$

which constitute part of the rules' premises in Tab. 4. Intuitively, they mean the following: label a is well-formed and well-typed wrt. the information Ξ and refers to an asynchronous call which results in a value of type T. If not interested in the type, we write $\Xi \vdash a : \mathit{ok}$, instead. The right-hand notation of (1) extends the binding context Ξ by the bindings transmitted as part of label a appropriately. For lack of space, we omit the formal definitions here. Intuitively, they make sure that only well-typed communication can occur and that the context is kept up-to date during reduction. Rule CallI deals with incoming calls, and basically adds the new thread n (which at the same time represents the future reference for the eventual result) in parallel with the rest of the program. In the configuration after the reduction step, the meta-mathematical notation $M.l(o)(\vec{v})$ stands for $t[o/s][\vec{v}/\vec{x}]$, when the method suite $[M]$ equals $[\ldots, l = \varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t, \ldots]$. Note that the step is only possible, if the lock of the object is free ($\bot$); after the step, the lock is taken ($\top$). Rule CallO deals with outgoing calls. Remember that

Table 3. Communication labels

$\gamma ::= n\langle \mathit{call}\ n.l(\vec{v})\rangle \mid n\langle \mathsf{return}(n)\rangle \mid \nu(n{:}T).\gamma$	basic labels
$a ::= \gamma? \mid \gamma!$	input and output labels

Table 4. External steps

$$\frac{a = \nu(\Xi').\, n\langle \mathit{call}\ o.l(\vec{v})\rangle? \qquad \Xi \vdash a : T \qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \parallel o[c, F, \bot] \xrightarrow{a} \acute{\Xi} \vdash C \parallel o[c, F, \top] \parallel n\langle \mathsf{let}\, x{:}T = M.l(o)(\vec{v})\ \mathsf{in\ release}(o); x\rangle} \ \textsc{CallI}$$

$$\frac{\begin{array}{c} a = \nu(\Xi').\, n\langle \mathit{call}\ o.l(\vec{v})\rangle! \quad \Xi' = \mathit{fn}(\lfloor a \rfloor) \cap \Xi_1 \\ \acute{\Xi}_1 = \Xi_1 \setminus \Xi' \quad \Delta \vdash o \quad \acute{\Xi} = \Xi + a \end{array}}{\Xi \vdash \nu(\Xi_1).(C \parallel n\langle \mathsf{let}\, x{:}T = o.l(\vec{v})\ \ \mathsf{in}\ \ t\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\acute{\Xi}_1).(C)} \ \textsc{CallO}$$

$$\frac{a = \nu(\Xi').\, n\langle \mathit{return}(v)\rangle? \qquad \Xi \vdash a : \mathit{ok} \qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash C \parallel n\langle v\rangle} \ \textsc{RetI}$$

$$\frac{a = \nu(\Xi').\, n\langle \mathit{return}(v)\rangle! \qquad \Xi' = \mathit{fn}(\lfloor a \rfloor) \cap \Xi_1 \quad \acute{\Xi}_1 = \Xi_1 \setminus \Xi' \quad \acute{\Xi} = \Xi + a}{\Xi \vdash \nu(\Xi_1).(C \parallel n\langle v\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\acute{\Xi}_1).C} \ \textsc{RetO}$$

an asynchronous call, as given in $\textsc{CallO}_i$ from Tab. 2, does not immediately lead to an interface interaction, but is an internal step, which only afterwards (asynchronously) leads to the interface interaction as specified in CallO. Thus the t in the consequence of the rule always equals stop and the named thread n serves only to issue the outgoing call. Furthermore, the binding context Ξ is updated and, additionally, previously private names mentioned in Ξ_1 might escape by scope extrusion, which is calculated by the second and third premise. Rules RetI and RetO deal with returning the value at the end of a method call.

We write $\Xi_1 \vdash C_1 \stackrel{t}{\Longrightarrow} \Xi_2 \vdash C_2$ if $\Xi_1 \vdash C$ reduces in a number of internal and external steps to $\Xi_2 \vdash C_2$, exhibiting t as the trace of the external steps.

3 A Behavioral Interface Specification Language

The behavior of an object (or a component consisting of a set of objects, for that matter) at the interface is described by a sequence of labels as given by Tab. 3. The black-box behavior of a component can therefore be described by a set of *traces*, each consisting of a finite sequence of labels. To specify sets of label traces, we employ a simple trace language with prefix, choice and recursion. Table 5 contains its syntax. The syntax of the labels in the specification language, naturally, quite resembles the labels of Tab. 3. Comparing Tabs. 3 and 5, there are two differences: first, instead of names or references n, the specification language here uses variables. Second, the labels here allow a binding of the form $(x{:}T).\gamma$, which has no analog in Tab. 3; the form $\nu(x{:}T).\gamma$ corresponds to $\nu(n{:}T).\gamma$, of course. Both binding constructs act as variable declarations, with the difference that $\nu(x{:}T).\gamma$ not just introduces a variable (together with its type T), but in

Table 5. Specification language

$\gamma ::= x\langle \mathit{call}\ x.l(\vec{x})\rangle \mid x\langle \mathsf{return}(x)\rangle \mid \nu(x{:}T).\gamma \mid (x{:}T).\gamma$	basic labels
$a ::= \gamma? \mid \gamma!$	input and output labels
$\varphi ::= X \mid \epsilon \mid a.\varphi \mid \varphi + \varphi \mid \mathsf{rec}\ X.\varphi$	specifications

addition asserts that the names represented by that variable must be fresh. The binding $(x{:}T).\gamma$ corresponds to a conventional variable declaration, introducing the variable x which represents arbitrary values (of type T), either fresh or already known.

In the specification, it is important to distinguish between input and output interactions, as input messages are under the control of the environment, whereas the outputs are to be provided by the object as specified. This splits the specification into an *assumption* part under the responsibility of the environment, and a commitment part, controlled by the component. Hence, the input interactions are the ones being *scheduled,* whereas the outputs are not; they are used for *testing* that the object behaves correctly. To specify non-deterministic behavior, the language supports a choice operator, and we distinguish between choices taken by the environment—external choice—and those the object is responsible for—internal choice. Especially, we do not allow so-called mixed choice, i.e., choices are either under control of the object itself and concerns outgoing communication, or under control of the environment and concerns incoming communication. These restrictions are formalized next as part of the well-formedness conditions.

3.1 Well-Formedness

The grammar given in Tab. 5 allows to specify sets of traces. Not all specifications, however, are meaningful, i.e., describe traces actually possible at the interface of a component. We therefore formalize conditions to rule out such ill-formed specification where the main restrictions are: *Typing:* Values handed over must correspond to the expected types for that methods. *Scoping:* Variables must be declared (together with their types) before their use. *Communication patterns:* No value can be returned before a matching outgoing call has been seen at the interface. Specifications adhering to these restrictions are called *well-formed.*

Well-formedness is given straightforwardly by structural induction by the rules of Tab. 6. The rules formalize a judgment of the form

$$\Xi \vdash \varphi : \mathit{wf}^{p} \tag{2}$$

which stipulates φ's well-formedness under the assumption context Ξ. The meta-variable p (for polarity) stands for either ?, !, or ?!, where ?! indicates the polarity for an empty sequence or for a process variable, and ? and ! indicate well-formed input and output specifications respectively. As before, Ξ contains bindings from variables and class names to their types. The class names are considered as constants and also, the context Ξ will remain unchanged during the well-formedness

Table 6. Well-formedness of trace specifications

$$\frac{}{\Xi \vdash \epsilon : wf^{?!}}\ \text{WF-E}\textsc{mtpy} \qquad \frac{\Xi \vdash X}{\Xi \vdash X : wf^{?!}}\ \text{WF-V}\textsc{ar}$$

$$\frac{a = \nu(\Xi').n\langle call\ o.l(\vec{v})\rangle? \qquad \Xi \vdash a : ok \qquad \acute{\Xi} = \Xi + a \qquad \acute{\Xi} \vdash \varphi : wf^{p}}{\Xi \vdash a.\varphi : wf^{?}}\ \text{WF-C}\textsc{all}\text{I}$$

$$\frac{a = \nu(\Xi').n\langle return(v)\rangle? \qquad \Xi \vdash a : ok \qquad \acute{\Xi} = \Xi + a \qquad \acute{\Xi} \vdash \varphi : wf^{p}}{\Xi \vdash a.\varphi : wf^{?}}\ \text{WF-R}\textsc{et}\text{I}$$

$$\frac{\Xi \vdash \varphi_1 : wf^{p} \qquad \Xi \vdash \varphi_2 : wf^{p}}{\Xi \vdash \varphi_1 + \varphi_2 : wf^{p}}\ \text{WF-C}\textsc{hoice} \qquad \frac{\Xi, X \vdash \varphi : wf^{p}}{\Xi \vdash \mathsf{rec}\ X.\varphi : wf^{p}}\ \text{WF-R}\textsc{ec}$$

derivation, since all classes are assumed to be known in advance and class names cannot be communicated. This is in contrast to the variables, which represent object references and references to future variables (resp. thread names). Besides that, the context also stores process variables X. The rules work as follows: The empty trace is well-formed (cf. rule WF-Empty), and a process variable X is well-formed, provided it had been declared before (written $\Xi \vdash X$, cf. rule WF-Var). We omit the rules WF-CallO and WF-RetO for outgoing calls, resp. outgoing get-labels, as they are dual to WF-CallI and WF-RetI.

3.2 Observational Blur

Creol objects communicate asynchronously and the order of messages might not be preserved during communication. Thus, an outside observer or tester can not see messages in the order in which they had been sent, and we need to relax the specification up-to some appropriate notion of *observational equivalence*, denoted by $\equiv_{obs}$ and defined by the rules of Tab. 7. Rule Eq-Switch captures the asynchronous nature of communication, in that the order of outgoing communication does not play a role. The definition corresponds to the one given in [7] and also of [8], in the context of multi-threading concurrency. Rule Eq-Plus allows to distribute an output over a non-deterministic choice, *provided* that it's a choice itself over outputs, as required by the well-formed condition in the premise. Rule Eq-Req finally expresses the standard unrolling of recursive definitions. We omit further standard equivalence rules, such as defining commutativity and associativity of $+$ and neutrality of ϵ.

Next we state that well-formedness is preserved under the given equivalence.

Lemma 1. *If $\Xi \vdash \varphi : wf^{p}$ and $\varphi \equiv_{obs} \varphi'$, then $\Xi \vdash \varphi' : wf^{p}$.*

Given the equivalence relation, the meaning of a specification is given operationally by the rather obvious reduction rules of Tab. 8. The next lemmas

Table 7. Observational equivalence

$$\frac{}{\nu(\Xi)\ .\ \gamma_1!\ .\ \gamma_2!\ .\ \varphi \equiv_{obs} \nu(\Xi)\ .\ \gamma_2!\ .\ \gamma_1!\ .\ \varphi}\ \textsc{Eq-Switch}$$

$$\frac{\vdash (\varphi_1 + \varphi_2) : wf^{!}}{\gamma!.(\varphi_1 + \varphi_2) \equiv_{obs} \gamma!.\varphi_1 + \gamma!.\varphi_2}\ \textsc{Eq-Plus} \qquad \mathsf{rec}\ X.\varphi \equiv_{obs} \varphi[\mathsf{rec}\ X.\varphi/X] \qquad \textsc{Eq-Rec}$$

Table 8. φ rules

$$\frac{\acute{\Xi} = \Xi + a}{\Xi \vdash a.\varphi \xrightarrow{a} \acute{\Xi} \vdash \varphi}\ \textsc{R-Pref} \qquad \frac{\Xi \vdash \varphi_1 \xrightarrow{a} \acute{\Xi} \vdash \varphi_1'}{\Xi \vdash \varphi_1 + \varphi_2 \xrightarrow{a} \acute{\Xi} \vdash \varphi_1'}\ \textsc{R-Plus}_1$$

$$\frac{\varphi \equiv_{obs} \varphi' \qquad \Xi \vdash \varphi' \xrightarrow{a} \Xi \vdash \varphi''}{\Xi \vdash \varphi \xrightarrow{a} \Xi \vdash \varphi''}\ \textsc{R-Equiv}$$

express simple properties of the well-formedness condition, connecting it to the reduction relation.

Lemma 2. *Assume $\Xi \vdash \varphi : wf$.*

1. *Exactly one of the three conditions holds: $\Xi \vdash \varphi : wf^{?!}$, $\Xi \vdash \varphi : wf^{?}$, or $\Xi \vdash \varphi : wf^{!}$*
2. *If $\varphi \xrightarrow{a}$ with a an input, then $\Xi \vdash \varphi : wf^{?}$. Dually for outputs.*
3. *If $\Xi \vdash \varphi : wf^{?}$, then $\varphi \xrightarrow{a}$ with a an input. Dually for outputs.*

Lemma 3 (Subject reduction). *$\Xi \vdash \varphi : wf$ and $\Xi \vdash \varphi \xrightarrow{a} \acute{\Xi} \vdash \acute{\varphi}$, then $\acute{\Xi} \vdash \acute{\varphi} : wf$.*

Lemma 4. *Assume $\Xi \vdash C$. If $\Xi \vdash C \xRightarrow{t}$, then $\Xi \vdash \varphi_t : wf$ (where φ_t is the trace t interpreted to conform to Tab. 5, i.e., the names of t are replaced by variables).*

4 Scheduling and Asynchronous Testing of Creol Objects

Next we put together the (external) behavior of an object (Sect. 2) and its intended behavior specified as in Sect. 3. Table 9 defines the interaction of the interface description with the component, basically by synchronous parallel composition. Both φ and the component must engage in corresponding steps, which, for incoming communication schedules the order of interactions with the component whereas for outgoing communication the interaction will take place only if it matches an outgoing label in the specification and an error is raised if input is required by the specification. The component can proceed on its own via

Table 9. Parallel composition

$$\frac{\Xi \vdash C \xrightarrow{\tau} \Xi \vdash \acute{C}}{\Xi \vdash C \parallel \varphi \to \Xi \vdash \acute{C} \parallel \varphi}\ \textsc{Par-Int}$$

$$\frac{\Xi_1 \vdash C \xrightarrow{a} \acute{\Xi}_1 \vdash \acute{C} \qquad \Xi_1 \vdash \varphi \xrightarrow{b} \acute{\Xi}_2 \vdash \acute{\varphi} \qquad \vdash a \lesssim_\sigma b}{\Xi_1 \vdash C \parallel \varphi \to \acute{\Xi}_1 \vdash \acute{C} \parallel \acute{\varphi}\sigma}\ \textsc{Par}$$

$$\frac{\Xi \vdash \varphi : wf^?}{\Xi \vdash \nu(\Xi').(C \parallel n\langle \mathsf{let}\, x{:}T = o.l(\vec{v})\ \mathsf{in}\ t\rangle \parallel \varphi) \to \lightning}\ \textsc{Par-Err-Call}$$

$$\frac{\Xi \vdash \varphi : wf^?}{\Xi \vdash \nu(\Xi').(C \parallel n\langle v\rangle \parallel \varphi) \to \lightning}\ \textsc{Par-Err-Ret}$$

internal steps (cf. rule Par-Int). Rule Par requires that, in order to proceed, the component and the specification must engage in the "same" step, where φ's step b is matched against the step a of the component. The matching is not simple pattern matching as it needs to take into account in particular the two different kinds of bindings in the specification language, $\nu(x{:}T)$ as the freshness assertion and $(x{:}T)$ representing standard variable declarations. Here $\vdash a \lesssim_\sigma b$ states that there exist a substitution σ such that the label a produced by the component and the label b specified by the interface description can be matched. We omit the details of the matching and refer to the longer version [6]. The rules Par-Err-Call and Par-Err-Ret report an error if the specification requires an input as the next step and the object however could do an output, either a call or a return. In the rule $\lightning$ indicates the occurrence of an error. Note that the equivalence relation, according to the rule Eq-Switch, allows the reordering of outputs, but not of inputs.

Example 1. To illustrate the testing we sketch the well-known example of a travel agency. A client asks the travel agent for a cheap flight and the travel agent finds the cheapest flight by asking the flight companies. To test an implementation of the travel agent program we give a specification modeling the behavior of the client and the flight companies and specifying the expected behavior of the travel agent. The client sends two messages. First a start message and then the request. The travel agent tries to get the price information from the flight companies and then reports the result to the client.

$$\begin{aligned}\varphi_b = {} & n_{c1}\langle call\ b.start()\rangle? \ .\ n_{c1}\langle return()\rangle! \ .\ n_{c2}\langle call\ b.getPrice(x)\rangle? \ .\\ & n_1\langle call\ p_1.l(x)\rangle! \ .\ n_2\langle call\ p_2.l(x)\rangle! \ .\\ & n_1\langle return(v_1)\rangle? \ .\ n_2\langle return(v_2)\rangle? \ .\ n_{c2}\langle return(min_v)\rangle!\end{aligned}$$

5 Implementing a Specification-Driven Creol Interpreter

The operational semantics of the object-oriented language Creol [1] is formalized in rewriting logic [9] and executable on the Maude rewriting engine [10]. To obtain a *specification-driven interpreter* for testing Creol objects, we have formalized our behavioral interface specification language in rewriting logic, too. In the combined implementation we synchronize communication between specification terms and objects. The specification generates the required input to the object and tests whether the output behaviour of the object conforms to the specification. The original Creol interpreter consists of 21 rewrite rules and the extension adds 20 more.

We have argued that specified method calls should not be placed into the callee's input queue, but the call should be answered immediately. I.e., if an incoming call is specified and the lock of the object is free, the corresponding method code should start executing immediately. In the current version of the interpreter the incoming messages are generated from the specification, which amounts to the same as only allowing scheduled calls to interact with the object.

A Creol state configuration is a multiset of objects, classes, and messages. The rewrite rules for state transitions are on the form `rl Cfg => Cfg'`, effectively evolving the state of one object by executing a statement. Some statements generate new messages. Finally, some rules are concerned with scheduling processes and receiving messages. For the scheduling interpreter we introduce terms `Spec` for specifications and add rules on the form `(Spec || O) Cfg => (Spec' || O') Cfg'` to test the object `O` with respect to `Spec`, where `||` represents the synchronous parallel composition. Each rule evolves the state of a specification and the state of an object in a synchronized manner: any interaction only takes place when it matches a complementary label in the specification. For example, the Par rule in Tab. 9 is implemented by several Maude rules, of which we show `Par-incoming-call` and `Par-remote-async-call`, that handle the cases of synchronized incoming and outgoing calls; we also show the `Par-Err-Call` rule in Tab. 10. The rules are conditional rewrite rules, in which conditions of the form `Var:=term` bind `term` to the variable `Var`. Parts of the term that are not changed, like attributes, are represented by "...".

The rule `Par-incoming-call` combines the R-Pref rule in Tab. 8 for the specification with the CallI rule in Tab. 4 for interface behavior via the Par rule. The rule only applies if the process, `Pr` of the object `<O:C | ...>` is `idle` (i.e., the lock is free). The specification for `O`, `<call(T,R,M,P)? . sp>(O)`, starts with an incoming call label with thread name `T`, receiver `R`, method name `M`, and parameters `P`, and could by R-Pref reduce to `sp`. The careful reader might expect that the receiver mentioned in the specification should be the same as the object identifier `O`. However since a specification can contain variables, the receiver `R` *might* be identical to `O` but it may also be a variable, which will be matched with `O` in the `procLab` function. The function `procLab` (process label) generates concrete values from the variables in the specification label; builds an *invoc* message, i.e. a term representing a method call; and returns the message and a mapping of the variables to the values. The message and the

Table 10. Sample Maude rules

```
crl <call(T,R,M,P)?.sp>(O) || <O:C | ..., Pr:idle, ...> Cfg
 => <app(getS(Res),sp)>(O) || <O:C | ..., Pr:synch, ...> getM(Res) Cfg
if Res:=procLab(O , call(T,R,M,P)?) [label Par-incoming-call] .

crl <call(T,R,M,P)!.sp>(O) || <O:C | ...,Pr:{L | call(A;E;Q;EL);SL},...> Cfg
 => <app(Sub,sp)>(O)       || <O:C | ...,Pr:{insert(A,Lab,L) | SL},...> Cfg
    (invoc(O,Lab,Q,Args) from O to Rcv)
if Lab:=label(O,N) ∧ Rcv:=evalGuard(E,(S::L),noMsg) ∧
   Args:=evalGuardList(EL,(S::L),noMsg) ∧
   Sub:=matchCall(Lab,Rcv,Q,Args,call(T,R,M,P)) ∧ noMismatch(Sub)
[label Par-remote-async-call] .

crl <inSp>(O)    || <O:C | ...,Pr:{L | call(A;E;Q;EL);SL},...> Cfg
 => <epsilon>(O) || <O:C | ...,Pr:{L | call(A;E;Q;EL);SL},...> Cfg
    errorMsg("ERROR") if E=/="this" [label Par-Err-Call] .
```

substitution are extracted by the functions `getM` and `getS`, respectively. The message is placed into the configuration and the substitution is applied to `sp` using the `app` function. Method binding and the rules for executing the bound code are specified by equations in the Creol interpreter. Since equations will be applied before any other rewrite rules this ensures that the execution of the code resulting from the call starts before any other invoc message can interfere.

In the `Par-remote-async-call` rule the object is in a state where the next step in the executing process is an outgoing call and the specification starts with a call out. The `matchCall` function tries to match the concrete values derived from the object's state against the variables in the label. The condition `noMismatch(Sub)` blocks the conditional rule if no match is possible, otherwise the outgoing call takes place and the substitution `Sub` is applied to the remainder of the specification. The last rule implements the Par-Err-Call rule. The distinction between input and output specifications is enforced by different subsorts: the variable `inSp` matches all specifications of incoming messages. When the next step of the executing process is a call statement, then this leads to an error, as expected.

Here we focus on the run-time behavior of specifications. Hence, we simply assume well-formedness and don't give the Maude formalization.

6 Conclusion

We have presented a formalization of the interface behavior of Creol together with a behavioral interface specification language. We have formally described how to use this specification language for black-box testing of asynchronously communicating Creol components and we have presented our rewriting logic implementation of the testing framework.

Related work. Systematic testing is indispensable to assure quality of software and systems (cf. [11,12,13,14,15], amongst others). [16] presents an approach to integrate black-box and white-box testing for object-oriented programs. Equivalence is based on the idea of observably equivalent terms and fundamental pairs as test cases, but not in an asynchronous setting (and as in [17] [18] [19] [20]). In the approach, pairs of (ground) terms are used for the test cases. Testing for *concurrent* object-oriented programs based on synchronization sequences is investigated in [21], using Petri nets and OBJ as foundation. Long in his thesis [22] presents ConAn ("concurrency analyser"), which generates test drivers from test scripts. The method allows to specify sequences of component method calls and the order in which the calls should be issued. It can be seen as an extension of the testing method for monitors from [23]. For scheduling the intended order, an external *clock* is used, which is introduced for the purpose of testing, only. In the context of $C^{\#}$, [24] presents model-based analysis and model-based testing, where abstract models of object-oriented programs are tested. The approach, however, does not target concurrent programs.

Even if not specifically targeting Creol, [25] pursues similar goals as this paper, validating component interfaces specified in rewriting logic. In contrast to here, the interface behavior is specified by first-order logic over traces, where from a given predicate an assumption part and a guarantee part can be derived. Our approach is more specific in that we schedule incoming calls to a component, and test the output behavior.

In [26], the authors target Creol as language and investigate how different schedulings of object activity restrict the behavior of a Creol object, thus leading to more specific test scenarios. The focus, however, is on the *intra*-object scheduling, and the test purposes are given as assertions on the *internal* state of the object. This is in contrast to the setting here, focusing on the interface communication. The testing methodologies are likewise different. We execute the behavioral trace specification directly in composition with the implementation being tested. They use a scheduling strategy and a model for an object implementation to generate test cases which then are used afterwards to test for compliance with an implemented Creol object.

Future work. We plan to extend the theory to components under test instead of single objects. This leads to complex scheduling policies and complex specifications. Furthermore, there are several interesting features of the Creol language which may be added, including first-class futures, promises, processor release points, inheritance and dynamic class updates. For the specification language we want to investigate how to extend it with assertion statements on labels, which leads to scheduling policies sensitive to the *values* in the communication labels. Natural further steps for the implementation are to extend it to include a check for well-formedness according to Tab. 6, and also to modify the matching algorithm to distinguish between fresh and already known names. The generation of Creol messages from specifications can also be made more sophisticated to achieve better test coverage. It is also interesting to combine the approach we describe here with model checking and abstraction. By using the built-in *search* functionality of Maude, model checking of invariants can be done easily. We plan to additionally use Maude's LTL model checker with our testing framework.

Acknowledgement. We thank Andreas Grüner for giving insight to the field of testing of (concurrent) object-oriented languages, the members of the PMA group for valuable feedback and the anonymous referees for insightful and constructive criticism.

References

1. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. Theoretical Computer Science 365(1-2), 23–66 (2006)
2. de Roever, W.P., de Boer, F.S., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods. Cambridge University Press, Cambridge (2001)
3. The Creol language, `http://heim.ifi.uio.no/creol`
4. Abadi, M., Cardelli, L.: A Theory of Objects. Monographs in Computer Science. Springer, Heidelberg (1996)
5. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, part I/II. Information and Computation 100, 1–77 (1992)
6. Grabe, I., Kyas, M., Steffen, M., Torjusen, A.B.: Executable interface specifications for testing asynchronous Creol components. Technical Report 375, University of Oslo, Dept. of Computer Science (July 2008)
7. Steffen, M.: Object-Connectivity and Observability for Class-Based, Object-Oriented Languages. Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel (July 2006)
8. Jeffrey, A., Rathke, J.: A fully abstract may testing semantics for concurrent objects. In: 17th Annual IEEE Symposium on Logic in Computer Science, pp. 101–112. IEEE Computer Society Press, Los Alamitos (2002)
9. Meseguer, J.: Conditional rewriting as a unified model of concurrency. Theoretical Computer Science 96, 73–155 (1992)
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The Maude 2.0 system. In Nieuwenhuis, R., ed.: RTA 2003. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 76–87. Springer, Heidelberg (2003)
11. Myers, G.J.: The Art of Software-Testing. John Wiley & Sons, New York (1979)
12. Patton, R.: Software Testing, 2nd edn. SAMS (July 2005)
13. Gaudel, M.C.: Testing can be formal, too. In: Mosses, P.D., Schwartzbach, M.I., Nielsen, M. (eds.) CAAP 1995, FASE 1995, and TAPSOFT 1995. LNCS, vol. 915, pp. 82–96. Springer, Heidelberg (1995)
14. Binder, R.V.: Testing Object-Oriented Systems, Models, Patterns, and Tools. Addison-Wesley, Reading (2000)
15. Bertolino, A.: Software testing research: Achievements, challenges, dreams. In: FOSE 2007: Future of Software Engineering, pp. 85–103. IEEE Computer Society Press, Los Alamitos (2007)
16. Chen, H.Y., Tse, T.H., Chan, F.T., Chen, T.Y.: In black and white: An integrated approach to class-level testing of object-oriented program. ACM Transactions of Software Engineering and Methodology 7(3), 250–295 (1998)

17. Bernot, G., Gaudel, M.C., Marre, B.: Software testing based on formal specifications. IEEE Software Engineering Journal 6(6), 387–405 (1991)
18. Doong, R.K., Frankl, P.G.: The ASTOOT approach to testing object-oriented programs. ACM Transactions on Software Engineering and Methodology 3(2), 101–130 (1994)
19. Doong, R.K., Frankl, P.G.: Case studies on testing object-oriented programs. In: TAV4: Proceedings of the symposium on Testing, analysis, and verification, pp. 165–177. ACM Press, New York (1991)
20. Frankl, P.G., Doong, R.K.: Tools for testing object-oriented programs. In: Proceedings of the 8th Pacific Northwest Conference on Software Quality, pp. 309–324 (1990)
21. Chen, H.Y., Sun, Y.X., Tse, T.H.: A strategy for selecting synchronization sequences to test concurrent object-oriented software. In: Proceedings of the 27th International Computer Software and Application Conference (COMPSAC 2003). IEEE Computer Society Press, Los Alamitos (2003)
22. Long, B.: Testing Concurrent Java Components. PhD thesis, University of Queensland (July 2005)
23. Brinch Hansen, P.: Reproducible testing of monitors. Software – Practice and Experience 8, 223–245 (1978)
24. Jacky, J., Veanes, M., Campbell, C., Schulte, W.: Model-Based Software Testing and Analysis with C#. Cambridge University Press, Cambridge (2008)
25. Johnsen, E.B., Owe, O., Torjusen, A.B.: Validating behavioral component interfaces in rewriting logic. Fundamenta Informaticae 82(4), 341–359 (2008)
26. Schlatte, R., Aichernig, B., de Boer, F., Griesmayer, A., Johnsen, E.B.: Testing concurrent objects with application-specific schedulers. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 319–333. Springer, Heidelberg (2008)

Compositional Strategy Mapping

Gregor Gössler

INRIA Grenoble – Rhône-Alpes, Pop Art team

Abstract. With the increasing complexity of embedded systems, coupled with the need for faster time-to-market and high confidence in the reliability of the product, design methods that ensure correctness by construction are, when available, the solution of choice. When dealing with open systems, the system behavior has to be considered in terms of strategies. In this paper, we are interested in a design flow supporting the refinement of strategies, rather than in computing a strategy by performing discrete controller synthesis on some given level of abstraction. We consider a platform-based design process consisting of successive mapping steps. The goal of each step is to map a strategy constructed so far onto a lower-level platform. The mapping is performed component-wise, using an abstraction of the environment of each component. We provide compositionality results ensuring that the refinement carries over to the global strategy, and illustrate the approach with examples.

1 Introduction

With the increasing complexity of embedded systems, coupled with the need for faster time-to-market and high confidence in the reliability of the product, design methods that ensure correctness by construction are, when available, the solution of choice. When dealing with reactive systems, which interact with their environment, the behavior of the system to be designed has to be considered in terms of *strategies*: can some desired behavior be enforced in spite of the — potentially non cooperative — environment?

Computing a strategy satisfying some property is expensive, and although modular and compositional discrete controller synthesis have been studied for some decades, this remains a hard problem. In particular, progress properties are notoriously more difficult to tackle compositionally than safety properties. Therefore, most modular approaches focus on safety properties, e.g., [8,13], or consider special cases. The interesting approach of [4], based on secure equilibria, is limited to two components. More closely related to our work, [5] defines a strong notion of modular refinement between "sociable" interface automata — allowing for interaction through shared actions and variables —, such that the refinement relation between a pair of components is independent of the environment. [3] defines refining contexts ensuring compositional refinement in a framework of components communicating through streams.

In this paper, we are interested in a design flow supporting the refinement of strategies, rather than in discrete controller synthesis performed on some given

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 340–354, 2010.

level of abstraction. We consider a platform-based design process consisting of successive mapping steps [12]. The goal of each step is to map a strategy σ constructed so far, which guarantees a desired behavior on a system S, onto a system S'. If the mapping succeeds, a refinement relation is automatically constructed, showing how σ can be implemented using the primitives provided by S'. The mapping is performed component-wise, using an abstraction of the behavior of the environment of each component. We provide compositionality results ensuring that the refinement carries over to the global strategy.

Mapping is particularly interesting in a heterogeneous framework supporting different models of interaction and execution, and thus, different ways of implementing a desired behavior. We therefore formulate the results in a subset of the heterogeneous component framework BIP [10,2].

To our knowledge, this is the first work on compositional mapping of strategies. In contrast to many approaches limited to safety, strategy refinement and mapping support arbitrary strategies. We expect compositionally verifying refinement of a strategy ensuring some property to be less pessimistic than compositionally synthesizing a strategy ensuring the same property "from scratch", especially for properties other than safety.

The paper is organized as follows. Section 2 introduces the component framework. Section 3 defines strategy refinement and mapping, and provides a set of compositionality, correctness, and completeness results. In Section 4 we illustrate the method with the distributed implementation of a centralized specification. Section 5 concludes.

2 Component Model

In the following, we present a simplified version of the component model BIP (behavior – interaction – priority) introduced in [10,1,2].

Given a relation $R \subseteq X \times Y$, let $dom(R) = \{x \in X \mid \exists y \in Y \,.\, (x, y) \in R\}$. For $X' \subseteq X$, let $R(X') = \{y \in Y \mid \exists x \in X' \,.\, (x, y) \in R\}$.

We consider action vocabularies A partitioned into *controllable* actions A^c and *uncontrollable* actions A^u. Uncontrollable actions are used to represent input events that cannot be triggered nor prevented by the modeled system.

Definition 1 (Interaction model). *An* interaction α *over an action vocabulary (or set of ports) A is a subset of A. An* interaction model *over A is a set IC of interactions such that $\bigcup_{\alpha \in IC} \alpha = A$.*

An interaction is a set of component actions taking place simultaneously. For the sake of simplicity, we suppose that for any interaction α, $\alpha \cap A^u \neq \emptyset \implies |\alpha| = 1$, that is, uncontrollable actions can only interleave. According to the presence of uncontrollable actions, we partition IC into IC^c and IC^u.

Definition 2 (Behavior). *A* behavior *is a tuple $B = (Q, IC, \rightarrow)$ with Q a set of states, IC an interaction model, and $\rightarrow \subseteq Q \times IC \times Q$ a deterministic transition relation (that is, $q \xrightarrow{\alpha} q' \wedge q \xrightarrow{\alpha} q'' \implies q' = q''$).*

Given a behavior $B = (Q, IC, \rightarrow)$, $\alpha \in IC$, and $q \in Q$, let $\mathsf{enabled}_B(\alpha)(q) \iff \exists q' \in Q \; . \; q \xrightarrow{\alpha} q'$.

Definition 3 (Predecessors). *Given a behavior $B = (Q, IC, \rightarrow)$ and a set of states $Q' \subseteq Q$, the* predecessors *of Q' by interaction α is the predicate $pre_\alpha(Q') = \{q \in Q \mid \exists q' \in Q' \; . \; q \xrightarrow{\alpha} q'\}$.*

We define two operations on behaviors: restriction and composition.

Given a behavior B_k with vocabulary A_k and an interaction α, let $\alpha[k] = \alpha \cap A_k$, and $IC[k] = \{\alpha[k] \mid \alpha \in IC\} \setminus \{\emptyset\}$.

Definition 4 (Composition). *Given an interaction model IC and a set of behaviors $B_i = (Q_i, IC[i], \rightarrow_i)$, $i = 1, ..., n$, with disjoint action vocabularies, let $\|_{IC}\{B_i\}_i = (Q, IC, \rightarrow)$ be their composition such that $Q = Q_1 \times ... \times Q_n$ and $q \xrightarrow{\alpha} q'$ if $\alpha \in IC$, and for any $i = 1, ..., n$, $q_i \xrightarrow{\alpha[i]}_i q'_i$ if $\alpha[i] \in IC[i]$, and $q'_i = q_i$ if $\alpha[i] = \emptyset$.*

This is the standard synchronized product with interactions IC. A higher-level definition of composition, allowing for composition of interaction models, can be found in [10].

Definition 5 (Execution model). *An* execution model *over $B = (Q, IC, \rightarrow)$ is a tuple of predicates $U = (V_\alpha)_{\alpha \in IC^c}$ over Q. The* restriction *of B with U is the behavior $B/U = (Q, IC, \rightarrow')$ where $\rightarrow' = \{(q, \alpha, q') \in \rightarrow \mid V_\alpha(q) \vee \alpha \in IC^u\}$.*

In this simplified presentation we call *component* a behavior, and *system* a top-level component. In a composition $\|_{IC}\{B_i\}_i$, we note an interaction α of some component B_k as $B_k.\alpha$, or α if there is no ambiguity.

Definition 6 (Interface interaction). *Given an interaction α and a component B_k with vocabulary A_k, let $IF_k(\alpha) = \alpha \setminus A_k$ be the* interface interaction *of α.*

3 Strategy Refinement

We first introduce some notions used to reason about strategies.

Definition 7 (Unfolding). *An unfolding of a behavior $B = (Q_B, IC, \rightarrow_B)$ is a tuple $\sigma = (Q, IC, \rightarrow, l, Q^0)$ such that $(Q, IC, \rightarrow)$ is a behavior, $l : Q \rightarrow Q_B$ is a total labeling function such that $\forall q, q' \in Q \; \forall \alpha \in IC \; . \; q \xrightarrow{\alpha} q' \implies l(q) \xrightarrow{\alpha}_B l(q')$, and $Q^0 \subseteq Q$ is a set of* initial states*. By abuse of notation, we identify B with its trivial unfolding $(Q_B, IC, \rightarrow_B, id_{Q_B}, Q_B)$.*

Example 1. Consider the system $mutex = \|_{IC}\{P_1, P_2, S\}$ consisting of the three components shown in Fig. 1 with $IC = \{a_1, p_1|p, v_1|v, a_2, p_2|p, v_2|v\}$. Two processes contend for the use of a shared resource, with a semaphore ensuring mutual exclusion. All actions are controllable.

Fig. 2 shows an unfolding σ of $mutex$ modeling a first-come-first-served policy for fair resource use. σ has two states labeled with the same product state $w_1 w_2$

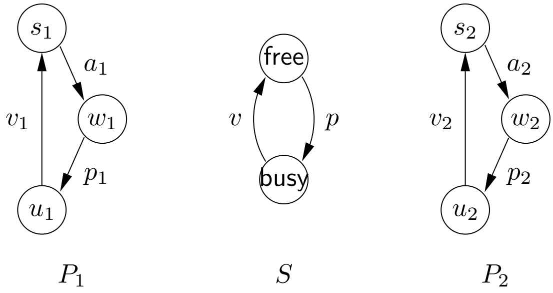


Fig. 1. Two components P_1 and P_2 coordinated by a semaphore S

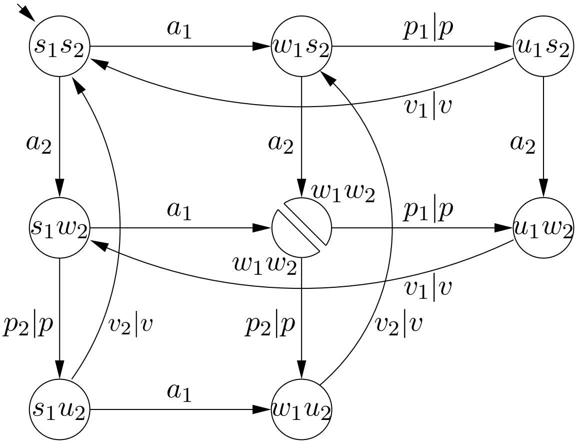


Fig. 2. Unfolding σ (the semaphore states are omitted)

where both components are waiting for access to the critical section. Depending on the order in which the access has been requested ($a_1; a_2$ or $a_2; a_1$), the component having made the request first is granted access.

We extend the definitions of composition and restriction to unfoldings.

Definition 8 (Composition). *Given an interaction model IC and unfoldings $\sigma_i = (Q_i, IC[i], \rightarrow_i, l_i, Q_i^0)$, $i = 1, ..., n$, let $\|_{IC}\{\sigma_i\}_i = (Q, IC, \rightarrow, l, Q^0)$ where*

- *$Q = Q_1 \times ... \times Q_n$, $Q^0 = Q_1^0 \times ... \times Q_n^0$;*
- *$q \xrightarrow{\alpha} q'$ if $\alpha \in IC$ and for any $i = 1, ..., n$, either $q_i \xrightarrow{\alpha[i]}_i q_i'$, or $\alpha[i] = \emptyset$ and $q_i' = q_i$.*
- *$l(q_1, ..., q_n) = (l_1(q_1), ..., l_n(q_n))$.*

Definition 9 (Restriction). *The* restriction *of an unfolding $\sigma = (Q, IC, \rightarrow, l, Q^0)$ with $U = (V_a)_{a \in IC^c}$, is the unfolding $\sigma/U = (Q, IC, \rightarrow', l, Q^0)$ where $\rightarrow' = \{(q, \alpha, q') \in \rightarrow \mid \alpha \in IC^u \vee V_\alpha(q)\}$.*

Definition 10 (Strategy). *A strategy σ over a behavior $B = (Q_B, IC, \rightarrow_B)$ is an unfolding $\sigma = (Q, IC, \rightarrow, l, Q^0)$ of B that is closed under uncontrollable transitions, that is, if $q \in Q$ and $\exists \alpha \in IC^u\ \exists y\ .\ l(q) \xrightarrow{\alpha}_B y$, then $\exists q' \in Q$ s.t. $q \xrightarrow{\alpha} q'$ and $l(q') = y$.*

Example 2. The unfolding σ of *mutex* from Fig. 2 is a strategy.

Definition 11 (Projection). *Given an unfolding $\sigma = (Q, IC, \rightarrow, l, Q^0)$ over $\|_{IC}\{C_i\}_i/U$, let $\approx_k \subseteq Q \times Q$ be the greatest fixpoint of*

$$\begin{aligned}\approx_k = \big\{(p,q) \mid \forall \alpha \in IC\ .\ \alpha[k] \neq \emptyset \implies & \\ & \big(\forall p' \in Q\ .\ \big(p \overset{T^*\alpha}{\rightarrow} p' \implies \exists q'\ .\ (q \overset{T^*\alpha}{\rightarrow} q' \wedge p' \approx_k q')\big) \wedge \\ & \forall q' \in Q\ .\ \big(q \overset{T^*\alpha}{\rightarrow} q' \implies \exists p'\ .\ (p \overset{T^*\alpha}{\rightarrow} p' \wedge p' \approx_k q')\big)\big)\big\}\end{aligned}$$

where $T = \{\alpha \in IC \mid \alpha[k] = \emptyset\}$. The projection$\pi_k(\sigma)$ *of σ on $C_k = (Q_k, IC[k], \rightarrow_k)$ is the unfolding $(Q', IC[k], \rightarrow', l', Q'_0)$ where $(Q', IC, \rightarrow', Q'_0)$ is the reduction of $(Q, IC, \rightarrow, Q^0)$ with respect to the weak bisimulation equivalence $\approx_k$ [7]. For $q \in Q$ we write $[q]_k$ for the state of $\pi_k(\sigma)$ representing the equivalence class of q. We put $l'([q]_k) = q_k$ if $l(q) = (q_1, ..., q_n)$.*

Example 3. The projection $\sigma_i = \pi_i(\sigma)$ of σ on each of the components P_1, P_2, and S, of Example 1 is equal to the trivial unfolding of the components.

Given a system $B = \|_{IC}\{B_i\}_{i \in K}/U = (Q, IC, \rightarrow)$, an unfolding $\sigma = (Q_\sigma, IC, \rightarrow_\sigma, l, Q^0_\sigma)$ over B, $\alpha \in IC$, and $q \in Q_\sigma$, let $\mathsf{enabled}_\sigma(\alpha)(q) \iff \exists q' \in Q_\sigma\ .\ q \xrightarrow{\alpha} q'$. For $k \in K$, let $\mathsf{disabled}_{k,\sigma}(\alpha)$ be a function associating with each interaction α a predicate over $Q \times Q_\sigma$ characterizing the states in which α is locally enabled in all involved components except for k, but disabled in σ:

$$\mathsf{disabled}_{k,\sigma}(\alpha) = \neg \mathsf{enabled}_\sigma(\alpha) \wedge \bigwedge_{\substack{i \neq k \\ \alpha[i] \neq \emptyset}} \mathsf{enabled}_{B_i}(\alpha[i])$$

We first define (non-compositional) strategy refinement.

Definition 12. *Given behaviors $B_1 = (Q_1, IC, \rightarrow_1)$ and $B_2 = (Q_2, IC', \rightarrow_2)$, unfoldings $\sigma = (Q, IC, \rightarrow, l, Q^0)$ and $\sigma' = (Q', IC', \rightarrow', l', Q'_0)$ over B_1 and B_2, respectively, and a relation $\sqsubseteq \subseteq Q_2 \times Q_1$, we define $\leq_{B_1,B_2}(\sigma, \sigma') \subseteq Q' \times Q$ as the greatest fixpoint of*

$$\begin{aligned}\prec = \sqsubseteq \cap \big\{(y,x) \mid \forall \alpha \in IC^c\ \ s.t.\ \ x \xrightarrow{\alpha} x' & \\ & \exists m \geq 0\ \exists b_0, ..., b_m \in (IC')^c\ \exists y_1, ..., y_m, y' \in Q'\ . \\ & y_0 = y \overset{b_0}{\rightarrow}{}' \ldots \overset{b_{m-1}}{\rightarrow}{}' y_m \overset{b_m}{\rightarrow}{}' y' \wedge \forall i = 1, ..., m\ .\ y_i \prec x \wedge y' \prec x' \qquad (1) \\ & \wedge \forall \beta \in (IC')^u\ .\ y \xrightarrow{\beta}_2 y'' \implies \\ & \qquad \big(y'' \prec x\ \vee\ \exists \alpha \in IC^u\ .\ x \xrightarrow{\alpha} x' \wedge y'' \prec x'\big)\big\} \qquad (2)\end{aligned}$$

σ' refines σ if $Q^0 \subseteq \leq_{B_1,B_2}(\sigma, \sigma')(Q'_0)$.

Intuitively, line (1) defines a simulation relation: from any pair of states x, y of the local strategies with $y \prec x$, whenever σ can make a transition $x \xrightarrow{\alpha} x'$, σ' can make a sequence of transitions such that both target states again satisfy the relation, and all intermediate states visited by σ' are related with x. Line (2) ensures refinement to be contravariant: simulation is preserved by all uncontrollable transitions of σ'. Unlike the usual definition of refinement with respect to safety properties, an unfolding σ' refines σ if it simulates σ (the underlying behaviors may be incomparable). This ensures that the behavior offered by σ — which may be used when the component B_1 is deployed — is also provided by σ'. The refinement relation is an *alternating simulation* [6]. A weaker definition of refinement such as (alternating) trace inclusion would not be sufficient to ensure compositionality in this framework.

3.1 Stability

The following definition adds sufficient conditions to Definition 12, ensuring refinement between two unfoldings σ and σ' to be compositional. Intuitively, the moves of σ' are required to be enabled in the global system *env*$'$ whenever the corresponding move of σ is enabled in *env*, and the effect of σ' on *env*$'$ must not be not more restrictive than the effect of σ on *env*. A more detailed explanation is given after the definition.

Definition 13 (Strategy refinement). *Given* $env = \big(\|_{IC}\{C_i\}_i\big)/U$, $env' = \big(\|_{IC'}\{C'_i\}_i\big)/U'$ *with state spaces* Q_{env} *and* $Q_{env'}$*, respectively, an unfolding* $\sigma_G = (Q_G, IC, \rightarrow_G, l_G, Q^0_G)$ *over env with projection* $\sigma = (Q, IC[k], \rightarrow, l, Q^0)$ *over* $C_k = (Q_k, IC[k], \rightarrow_k)$*, an unfolding* $\sigma' = (Q', IC'[k], \rightarrow', l', Q'_0)$ *over* $C'_k = (Q'_k,\ IC'[k], \rightarrow'_k)$*, and a predicate inv over* $Q_{env} \times Q_G \times Q_{env'}$ *called* refinement invariant*. We define* $\leq^{inv}_{env,env'} (\sigma, \sigma') \subseteq Q' \times Q$ *as the greatest fixpoint of*

$$
\begin{aligned}
\prec = \big\{(y, x) \in Q' \times Q \mid \forall \alpha \in IC^c \ \ s.t.\ \ x \overset{\alpha[k]}{\rightarrow} x' \\
&\exists m \geq 0\ \exists b_0, ..., b_{m-1} \in (IC')^c\ \exists \beta \in (IC')^c \cup \{\emptyset\}\ \exists y_1, ..., y_m, y' \in Q'\ . \\
&y_0 = y \overset{b_0}{\rightarrow}{}' \ldots \overset{b_{m-1}}{\rightarrow}{}' y_m \overset{b_m = \beta[k]}{\rightarrow}{}' y' \wedge IF_k(\alpha) = IF_k(\beta) \wedge y' \prec x' \wedge && (3)\\
&\forall i = 0, ..., m\ .\ \big(inv \wedge \mathsf{enabled}_{\sigma_G}(\alpha) \implies \\
&\qquad\qquad \mathsf{enabled}_{env'}(b_i)\big)[l(x)/Q_k, l'(y_i)/Q'_k] \wedge && (4)\\
&\forall i = 1, ..., m\ .\ y_i \prec x \wedge (y \sqsubseteq x \implies y_i \sqsubseteq x) \wedge && (5)\\
&\big(y_m \sqsubseteq x \implies pre_\alpha \circ pre_\beta(y' \sqsubseteq x')\big) && (6)\\
&\wedge \forall \beta \in (IC'[k])^u\ .\ \big(y \overset{\beta}{\rightarrow}{}'_k y'' \implies \big(y'' \prec x \wedge (y \sqsubseteq x \implies y'' \sqsubseteq x) \\
&\qquad \vee\ \exists \alpha \in IC[k]^u\ .\ x \overset{\alpha}{\rightarrow} x' \wedge y'' \prec x' \wedge (y \sqsubseteq x \implies y'' \sqsubseteq x')\big)\big)\big\} && (7)
\end{aligned}
$$

where we suppose w.l.o.g. that $\forall q \in Q'\ .\ q \overset{\emptyset}{\rightarrow}{}' q$*, and the predicate* $\sqsubseteq$ *is defined such that for any* $x \in Q_k$*,* $y \in Q'_k$*,*

$$
y \sqsubseteq x \iff inv \wedge \bigwedge_{\gamma \in IC^c \setminus IC[k]} \big(\mathsf{enabled}_{\sigma_G}(\gamma)[l(x)/Q_k] \implies \neg \mathsf{disabled}_{k,env'}(\gamma)[l'(y)/Q'_k]\big)
$$

σ' refines σ under inv, written $\sigma' \preceq^{inv}_{env,env'} \sigma$, if (1) $Q^0 \subseteq \leq (Q'_0)$, and (2) $\forall q_G \in Q^0_G \; \forall q \in Q^0 \; \forall q' \in Q' \, . \, (q' \leq q \implies \sqsubseteq [l_G(q_G)/Q_{env}, q_G/Q_G, \, l'(q')/Q'_k] = true)$, where $\leq = \leq^{inv}_{env,env'} (\sigma, \sigma')$.

σ' refines σ, written $\sigma' \preceq_{env,env'} \sigma$, if $\sigma' \preceq^{inv}_{env,env'} \sigma$ for some inv.

The definition of $\prec$ deserves some explanation. Line (3) states that for each transition α of σ, σ' can make a sequence of internal transitions, followed by a transition β such that $IF(\alpha) = IF(\beta)$, that is, both interactions are not distinguishable by other components than k. The target states must satisfy $\prec$ again. Line (4) requires that sequence of transitions to be enabled from y whenever α is enabled form x. Line (5) specifies that all intermediate states must refine x. The predicate $\sqsubseteq$ ("less restrictive than") over $Q_G \times Q_{env} \times Q_{env'}$ characterizes the states for which the behavior of other components than k is not more restricted in env' than in σ_G, both due to interactions that are not offered by k, or due to the execution model. In the sequel we will write $\sqsubseteq_{C_k}$ whenever an ambiguity may arise. The implication in line (5) ensures that $\sqsubseteq$ is invariant under the internal moves of σ'. Similarly, line (6) requires invariance of $\sqsubseteq$ under the pair of transitions α and β (both transitions act on disjoint state spaces, therefore pre_α and pre_β commute). Line (7) ensures contravariance: each uncontrollable transition of env' either preserves $\prec$ and $\sqsubseteq$, or is matched by an uncontrollable transition in σ such that the target states satisfy again $\prec$ while ensuring invariance of $\sqsubseteq$.

The relation $\leq$ depends on env and env'. This is because strategy refinement takes into account an abstraction of the environment, even if the actual simulation relation is local to a pair of components. The abstraction distinguishes states according to the interactions they disable; transitions are distinguished according to the set of interactions they can participate in.

The global unfolding σ_G allows to use information about the desired global behavior, if available, to make the definitions of $\mathsf{enabled}$ and $\sqsubseteq$ less pessimistic. Whenever the global strategy σ_G is not explicitly specified, we conservatively assume $\sigma_G = \big(\|_{IC}(\{\sigma\} \cup \{C_i\}_{i \neq k})\big)/U$. We drop the subscript env, env' when both systems are clear from the context.

The role of the refinement invariant is to provide information about related states in both systems env and env', e.g., to require equivalence of the states of two observers. In the simplest case, $inv = true$. Condition (1) in the definition of $\preceq$ ensures that every initial state of σ is simulated by some initial state of σ'; condition (2) ensures that $\sqsubseteq$ is satisfied for all initial states of σ_G.

Example 4. Consider the system $S = \|_{IC}\{\mathsf{initiator}, \mathsf{bus}, \mathsf{target}\}/U$ composed of the three components shown in Fig. 3, under the interaction model $IC = \{\mathsf{arrive}, \mathsf{!req}|\mathsf{?req}, \mathsf{timeout}, \mathsf{!ack}|\mathsf{?ack}, \mathsf{!msg}|\mathsf{?msg}, \mathsf{go}, \mathsf{done}\}$ and the restriction $U = (V_\alpha)_{\alpha \in IC}$, where $V_{\mathsf{arrive}} = \neg target.exec$, $V_{\mathsf{go}} = \neg initiator.exec$, and $V_\alpha = true$ otherwise. U thus ensures mutual exclusion between the gray component states, that is, invariance of the predicate $\neg(\mathsf{initiator.exec} \wedge \mathsf{target.exec})$. All actions except for $\mathsf{timeout}$ are controllable. Let us check whether (the trivial unfoldings of) the three components in S are refined by (the trivial unfoldings of) the components of Fig. 4 in $S' = \|_{IC'}\{\mathsf{initiator}', \mathsf{bus}', \mathsf{target}'\}/U$ with $IC' = (IC \cup \{\mathsf{lose}\}) \setminus \{\mathsf{timeout}\}$. We compute $\sqsubseteq_{\mathsf{initiator}}=$

$$
\begin{aligned}
&= inv \wedge \big(\mathsf{enabled}_{\mathsf{initiator}}(\mathsf{target.go}) \implies \neg\mathsf{disabled}_{\mathsf{initiator}'}(\mathsf{target}'.\mathsf{go})\big)\\
&\qquad \wedge \big(\mathsf{enabled}_{\mathsf{initiator}}(!\mathsf{req}|?\mathsf{req}) \implies \neg\mathsf{disabled}_{\mathsf{initiator}'}(!\mathsf{req}|?\mathsf{req})\big) \wedge \ldots\\
&= inv \wedge \big(\mathsf{target.waiting} \wedge \neg\mathsf{initiator.exec} \implies \neg(\mathsf{target}'.\mathsf{waiting} \wedge \mathsf{initiator}'.\mathsf{exec})\big)\\
&\qquad \wedge \big((\mathsf{initiator.exec} \vee \mathsf{initiator.retry}) \wedge \mathsf{bus.idle} \implies\\
&\qquad\qquad \mathsf{initiator}'.\mathsf{exec} \vee \mathsf{bus}'.\mathsf{busy}' \vee \mathsf{bus}'.\mathsf{done}'\big)
\end{aligned}
$$

and similarly for the two other pairs, and fix the refinement invariant to express that for any pair of initial and refining component, states of the same name are intended to be equivalent: $inv = (\mathsf{initiator.idle} \iff \mathsf{initiator}'.\mathsf{idle}) \wedge \ldots$. We compute the relation $\leq^{inv}_{S,S'}$ for each pair of components with the result shown in Table 1. In particular, the idle state of components initiator, bus, and target is refined by the idle state of initiator', bus', and target', respectively. Component target' refines target in spite of the added uncontrollable transition lose, since the target state idle of the latter refines state waiting of component target.

Assuming $\{\mathbf{idle}\}$ with **idle**=(idle,idle,idle) as the set of initial states of the strategy σ_G of S, we have $\sqsubseteq_{\mathsf{initiator}} [\mathbf{idle}/Q_S, \mathsf{initiator}'.\mathsf{idle}/Q_{\mathsf{initiator}'}] = true$. It

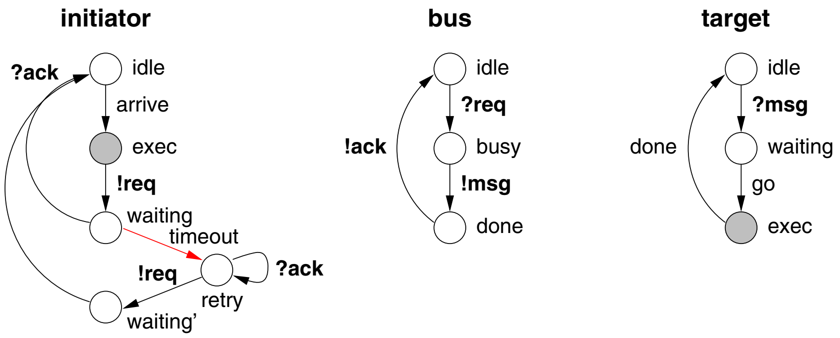


Fig. 3. Component behaviors

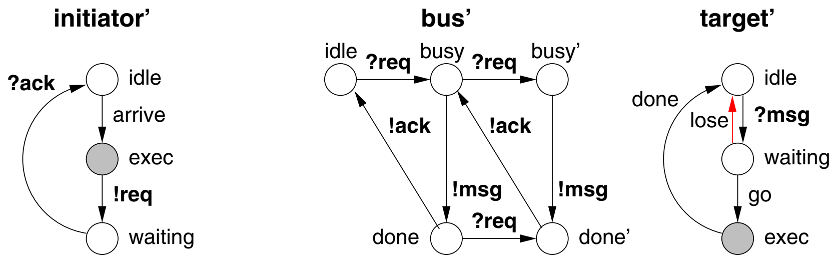


Fig. 4. Refining components

Table 1. Relations $\leq^{\mathsf{inv}}_{\mathsf{S,S'}}$ $(\mathsf{initiator}, \mathsf{initiator'})$, $\leq^{\mathsf{inv}}_{\mathsf{S,S'}}$ $(\mathsf{bus}, \mathsf{bus'})$, $\leq^{\mathsf{inv}}_{\mathsf{S,S'}}$ $(\mathsf{target}, \mathsf{target'})$

initiator	idle	exec	waiting	retry	waiting'
initiator'	idle	exec	waiting	-	waiting

target	idle	waiting	exec
target'	idle	idle, waiting	idle, exec

bus	idle	busy	done
bus'	idle, busy	busy, busy'	done, done'

follows that $\mathsf{initiator'} \preceq^{inv}_{S,S'} \mathsf{initiator}$. In the same way, we obtain $\mathsf{bus'} \preceq^{inv}_{S,S'} \mathsf{bus}$ and $\mathsf{target'} \preceq^{inv}_{S,S'} \mathsf{target}$.

Strategy refinement is compositional:

Theorem 1 (Strategy refinement). *Let* $env = \big(\|_{IC}\{C_i\}_i\big)/U$ *and* $env' = \big(\|_{IC'}\{C'_i\}_i\big)/U'$. *Given unfoldings* σ_i *over* C_i, σ'_i *over* C'_i, $i = 1, ..., n$, *and* σ_G *over* env, *if* $\forall i \;.\; \sigma'_i \preceq^{inv} \sigma_i$ *with respect to* σ_G *then* $\sigma' \preceq^{inv} \sigma_G$, *where* $\sigma' = \|_{IC'}\{\sigma'_i\}_i/U'$.

That is, the composition of locally refining strategies is a refining strategy.

Proof (sketch). Let $\leq_i = \leq^{inv}_{env,env'} (\sigma_i, \sigma'_i)$, $\sigma_G = (Q, IC, \rightarrow, l, Q^0)$, and $\sigma' = (Q', IC', \rightarrow', l', Q'_0)$. By hypothesis, $\forall k \;.\; \sigma'_k \preceq \sigma_k$. Assume that for some states $\mathbf{x} = (x_1, ..., x_n) \in Q$ and $\mathbf{y} = (y_1, ..., y_n) \in Q'$, $\forall i \;.\; y_i \leq_i x_i$. For $\alpha \in IC$, let $owners(\alpha) = \{k \mid \alpha[k] \neq \emptyset\}$.

- If $\mathbf{x} \xrightarrow{\alpha} \mathbf{x}' \in \sigma_G$, then by refinement between the local strategies there are $b_0, ..., b_{m-1}$ and $\beta \in (IC')^c \cup \{\emptyset\}$ with $owners(\alpha) = owners(\beta)$ enabled at $\mathbf{y}$ in σ' such that $y_i \xrightarrow{b_0} ... \xrightarrow{\beta[i]} y'_i \in \sigma'_i$ for $\beta[i] \neq \emptyset$ and $y_i = y'_i$ for $\beta[i] = \emptyset$, and $y'_i \leq_i x'_i$.
- By Definition 13, if $y_i \xrightarrow{u} y'_i$ then $y'_i \leq_i x_i$, or $x_i \xrightarrow{u} x'_i$ and $y'_i \leq_i x'_i$.

In both cases, $\sqsubseteq_i$ is preserved for $i = 1, ..., n$ by definition of $\leq$.

By induction it follows that $\mathbf{y} \leq \mathbf{x}$, where $\leq = \leq^{inv}_{env,env'} (\sigma_G, \sigma')$. By hypothesis, for any $q_i \in Q^0_i$ there exists $q'_i \in (Q^0_i)'$ such that $q'_i \leq q_i$, $i = 1, ..., n$. It follows that $Q^0 \subseteq \leq (Q'_0)$. On the global level, $\sqsubseteq_{env} = inv$, and by hypothesis we have $inv[l_G(q^0_G)/Q_{env}, q^0_G/Q_G, l'(q')/Q'] = true$ for any $q^0_G \in Q^0_G$, $q^0 \in Q^0$, and $q' \in Q'$ with $q' \leq q^0$. It follows that $\sigma' \preceq \sigma_G$.

Remark 1. Definition 13 and Theorem 1 assume one-to-one component refinement, in order to keep the syntactic simplicity of one-to-one correspondence. It is still possible to reason about refinement where components are removed or added, by representing a component C_k that exists only in S (resp. S'), by a "neutral" component $\bar{C}_k$ in S' (resp. S) with the same controllable interactions $IC[k]^c$ that are always enabled.

Definition 14 (Stability). *An unfolding p is* stable *in S if there exists a strategy σ such that $\sigma \preceq p$.*

Corollary 1 (Stability). *Given an unfolding p over $\|_{IC}\{C_i\}_i/U$ and a refinement invariant inv, if for each component C_k, $p_k = \pi_k(p)$ is stable — say, $\sigma_k \preceq^{inv} p_k$ for some strategy σ_k — then p is stable, and $\|_{IC}\{\sigma_i\}_i/U \preceq^{inv} p$.*

Proof. By Theorem 1, $\|_{IC}\{\sigma_i\}_i/U \preceq^{inv} \|_{IC}\{p_i\}_i/U$. Clearly, p is a restriction of $\|_{IC}\{p_i\}_i/U$. The claim $\|_{IC}\{\sigma_i\}_i/U \preceq^{inv} p$ follows from the fact that restricting the mapped unfolding strengthens the premises of the implications in Definition 13. ■

Example 5. Coming back to Example 4, suppose that we have found the unfolding σ shown in Fig. 5 cycling through the initial state (idle,idle,idle) of S. Notice that this unfolding is not a strategy, as it does not contain the uncontrollable timeout transition enabled in initiator.waiting. We want to find a strategy σ' over $S' = \|_{IC'}\{\mathsf{initiator}', \mathsf{bus}', \mathsf{target}'\}/U$.

We will use Corollary 1 to construct a global strategy σ' refining σ. We compute the local projections σ_i of σ on the components, as shown in Fig. 6 and check their refinement by the strategies given by the trivial unfoldings of the components of S'. It is not difficult to show that $\sigma'_i \preceq^{inv}_{S,S'} \sigma_i$ for all three components. According to Corollary 1, the trivial unfolding of S' refines σ.

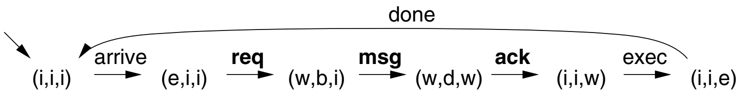


Fig. 5. Unfolding σ (where req =!req|?req etc.)

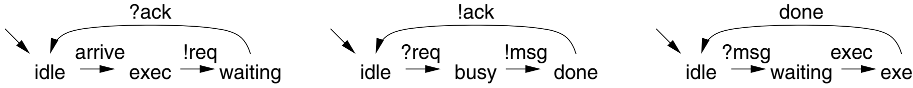


Fig. 6. Unfoldings $\sigma_i = \pi_i(\sigma)$

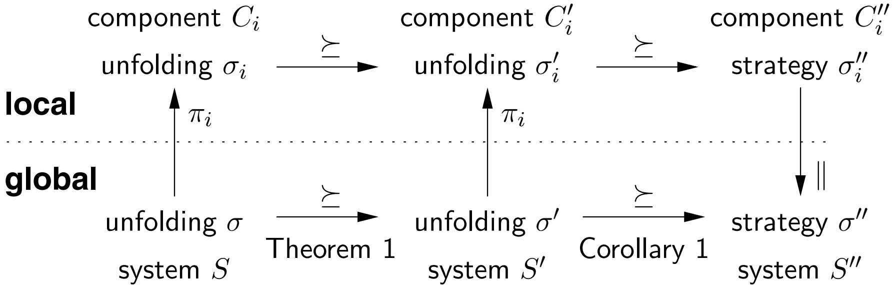


Fig. 7. Design flow using strategy refinement

The scheme of Fig. 7 summarizes a possible design flow supported by the results above.

3.2 Strategy Mapping

Theorem 1 provides a means to check for refinement between a pair of unfoldings. A strategy can be automatically and compositionally mapped on another system. The following definition and proposition allow to map a strategy σ on an unfolding σ', that is, construct the maximal sub-strategy $\sigma'' \subseteq \sigma'$ effectively refining σ.

Definition 15 (Strategy mapping). *Let $env = \big(\|_{IC}\{C_i\}_i\big)/U$, $env' = \big(\|_{IC'}\{C'_i\}_i\big)/U'$ with state spaces Q_{env} and $Q_{env'}$, respectively, an unfolding σ_G over env with projection $\sigma_k = (Q, IC[k], \rightarrow, l, Q^0)$ over $C_k = (Q_k, IC[k], \rightarrow_k)$, a component $C'_k = (Q'_k, IC'[k], \rightarrow'_k)$, an unfolding $\sigma'_k = (Q', IC'[k], \rightarrow', l', Q'_0)$ over C'_k, and a refinement invariant inv over $Q_{env} \times Q_G \times Q_{env'}$. The* mapping of σ_k on σ'_k, *written $\sigma_k \searrow_{inv} \sigma'_k$, is the unfolding $\sigma''_k = \big(dom(\leq), IC'[k], \rightarrow'', l', Q'_0 \cap dom(\leq)\big)$ where $\leq = \leq^{inv}_{env,env'} (\sigma_k, \sigma'_k)$ and*

$$\begin{aligned}\rightarrow'' = \Big\{(q_0, b, q_1) \in \rightarrow' \mid\ & b \in (IC')^u \ \vee\ \exists m \geq 0\ \exists b_1, ..., b_{m-1} \in (IC')^c\ \exists \alpha \in IC^c \\ & \exists \beta \in (IC')^c\ \exists q_2, ..., q_{m+1} \in Q'\ \exists \bar{q_0} \in \cap_{i=0,...,m} \leq (q_i)\ \exists \bar{q}_{m+1} \in \leq (q_{m+1})\ . \\ & q_0 \overset{b_0=b}{\rightarrow}{}' \dots \overset{b_m=\beta[k]}{\rightarrow}{}' q_{m+1} \wedge \bar{q_0} \overset{\alpha[k]}{\rightarrow} \bar{q}_{m+1} \wedge \beta[k] \neq \emptyset \wedge IF_k(\alpha) = IF_k(\beta) \wedge \\ & \forall i = 0, ..., m\ .\ \Big(\big(inv \wedge \mathsf{enabled}_{\sigma_G}(\alpha) \implies \\ & \qquad \mathsf{enabled}_{env'}(b_i)\big)[l(\bar{q_0})/Q_k, l'(q_i)/Q'_k] \wedge (q_0 \sqsubseteq \bar{q_0} \implies q_i \sqsubseteq \bar{q_0})\Big) \wedge \\ & \big(q_m \sqsubseteq \bar{q_0} \implies pre_\alpha \circ pre_\beta(q_{m+1} \sqsubseteq \bar{q}_{m+1})\big)\Big\}\end{aligned}$$

Proposition 1. *If σ'_k is a strategy then $\sigma_k \searrow_{inv} \sigma'_k$ (as defined in Definition 15) is a strategy.*

Proof. Since σ'_k is a strategy, it is closed under uncontrollable actions. By Definition 15, all uncontrollable actions of the mapping target are preserved in the mapped unfolding. ■

Mapping an unfolding σ on a target unfolding σ', rather than directly on the underlying behavior B, allows to improve precision, as mapped states may be distinguished in a mapping on σ' that may be confused when mapping on B.

Example 6. In Example 5, we verified refinement between the unfoldings σ_i and strategies σ'_i, where the latter were given. For components initiator and target, the same strategies can be obtained as the mappings $\sigma_{\mathsf{initiator}} \searrow_{inv} \mathsf{initiator}'$ and $\sigma_{\mathsf{target}} \searrow_{inv} \mathsf{target}'$, respectively. For component bus, the mapped strategy $\sigma_{\mathsf{bus}} \searrow_{inv} \mathsf{bus}'$ is the trivial unfolding of bus' from which the transition ?req from state done is removed.

Proposition 2 (Correctness). *Let $env = \big(\|_{IC}\{C_i\}_i\big)/U$, $env' = \big(\|_{IC'}\{C'_i\}_i\big)/U'$, $\sigma_i = (Q_i, IC[i], \rightarrow_i, l_i, Q^0_i)$ be an unfolding on C_i, $i = 1, ..., n$, and inv be a refinement invariant. If $\forall i\ .\ C'_i \preceq^{inv}_{env,env'} \sigma_i$, then*

- $\forall k \,.\, \sigma_k \searrow_{inv} C'_k \preceq^{inv} \sigma_k$, *and*
- $\|_{IC'}\{\sigma_i \searrow_{inv} C'_i\}_i/U' \preceq^{inv} \|_{IC}\{\sigma_i\}_i/U$.

Proposition 3 (Completeness). *If* $(\sigma \searrow_{inv} C') \not\preceq^{inv} \sigma$*, then there is no strategy* σ' *over* C' *refining* σ *under inv.*

Proof (sketch). Suppose on the contrary that such a σ' exists. Let $\sigma = (Q, IC, \rightarrow, l, Q^0)$, $C' = (Q', IC', \rightarrow')$, $\sigma' = (Q'', IC', \rightarrow'', Q''_0)$, $\leq_0 = \leq^{inv}_{env,env'} (\sigma, C')$, and $\leq_1 = \leq^{inv}_{env,env'} (\sigma, \sigma')$. There is a state $q_0 \in Q^0$ such that $q_0 \in \leq_1 (Q''_0) \setminus \leq_0 (Q')$ with some $\bar{q_0} \leq_1 q_0$. By definition of $\leq.$, there is some state q reachable from q_0 in σ and $\bar{q}$ reachable from $\bar{q_0}$ in σ' such that $\bar{q} \leq_1 q$ and

- either, some controllable successor simulating a state of σ exists in σ' but not in C'. Formally, $\exists q' \in Q \; \exists \alpha \in IC^c \,.\, q \xrightarrow{\alpha} q'$, and $\exists \bar{q}' \in Q'' \; \exists \beta \in (IC')^c \,.\, \bar{q} \xrightarrow{\beta}{}'' \bar{q}'$ with $\bar{q}' \leq_1 q'$. Then, $l(\bar{q}) \xrightarrow{\beta}{}' l(\bar{q}')$ (since σ' is an unfolding over C'), in contradiction to the hypothesis;
- or an uncontrollable successor of $l(\bar{q})$ not refining q or a successor, exists in C' but not in σ'. Formally, $\exists y \in Q' \; \exists \beta \in (IC')^u \,.\, l(\bar{q}) \xrightarrow{\beta}{}' y$. Then, $\exists \bar{q}' \in Q'' \,.\, \bar{q} \xrightarrow{\beta}{}'' \bar{q}'$ with $l(\bar{q}') = y$ (since σ' is a strategy, and C' is deterministic), which is again in contradiction to the hypothesis.

The claim follows.

4 Application: Distributed Algorithms

An interesting application domain of strategy mapping is to solve the following problem: how to deploy a centralized component-based system, whose components are coordinated through synchronization primitives, on a distributed platform where only lower-level communication primitives are available? We illustrate the principle with the mapping of a simple strategy ensuring mutual exclusion and fairness, on Kessels' distributed mutual exclusion algorithm [11].

Consider the centralized system $mutex = \|_{IC}\{P_1, P_2, S\}$ of Fig. 1 with $IC = \{a_1, p_1|p, v_1|v, a_2, p_2|p, v_2|v\}$ and the first-come-first-served strategy σ of Fig. 2. We call ww_1 (resp. ww_2) the state labeled with w_1w_2 reached with $a_1; a_2$ (resp. $a_2; a_1$) in which P_1 (resp. P_2) is served first. All other states of σ are uniquely identified by their label. Let σ_{P_1}, σ_{P_2}, and σ_S be the projections of σ on P_1, P_2, and S, respectively. We want to map strategy σ on the target platform given by two components implementing Kessels' mutual exclusion algorithm [11]:

```
K1:
req[1] = 1;
turn[1] = turn[2];
await (!req[2] or turn[1]!=turn[2]);
// critical section
req[1] = 0;
```

```
K2:
req[2] = 1;
turn[2] = !turn[2];
await (!req[1] or turn[1]=turn[2]);
// critical section
req[2] = 0;
```

According to Remark 1, we model Kessels' algorithm with two components and one dummy component $obs = \big(\{q_\bot\}, \{p, v\}, \big\{(q_\bot, \{p\}, q_\bot), (q_\bot, \{v\}, q_\bot)\big\}\big)$ "refining" the semaphore. *obs* has only one state and is always ready to make a p or v action. We will use this component as an observer to identify transitions in which one of the components K_1, K_2 enters or leaves its critical section.

Kessels' algorithm is modeled as component model $ks = \|_{IC'}\{K_1, K_2, obs\}/U$ with K_1 and K_2 as shown in Fig. 8, $IC' = \{t_1, t_1', a_1, p_1|p, v_1|v, t_2, t_2'a_2, p_2|p, v_2|v\}$ and $U = (V_\alpha)_{\alpha \in IC}$ defined as follows:

α	V_α
t_1	$turn_2$
t_1'	$\neg turn_2$
$p_1\|p$	$\neg req_2 \vee turn_1 \neq turn_2$
t_2	$\neg turn_1$
t_2'	$turn_1$
$p_2\|p$	$\neg req_1 \vee turn_2 = turn_1$

and $V_\alpha = true$ for all other $\alpha \in IC'$, where $req_i = l_i^2 \vee l_i^3 \vee l_i^4 \vee l_i^6 \vee l_i^7 \vee l_i^8$, and $turn_i = l_i^5 \vee l_i^6 \vee l_i^7 \vee l_i^8$. While the BIP framework supports variable assignment between components [1,9], the component states are explicit in the simplified presentation adopted here. We therefore represent the reading with a pair of transitions t_i and t_i' and the execution model shown above.

Let us now map strategy σ on ks, that is, implement mutual exclusion with the fairness constraint modeled by σ, in a distributed setting where the components K_1, K_2 communicate only through shared variables. According to Kessels' algorithm where in case of conflict, K_1 is given priority over K_2 if and only if $turn_1 \neq turn_2$, we fix the refinement invariant

$$inv = (ww_1 \implies turn_1 \neq turn_2) \wedge (ww_2 \implies turn_1 = turn_2)$$

and compute

$$\begin{aligned} \sqsubseteq_{P_1} = inv \ \wedge \ &\big(\mathsf{enabled}_{P_1,\sigma}(a_2) \implies \neg\mathsf{disabled}_{K_1,ks}(a_2)\big) \wedge \\ &\big(\mathsf{enabled}_{P_1,\sigma}(p_2|p) \implies \neg\mathsf{disabled}_{K_1,ks}(p_2|p)\big) \wedge \\ &\big(\mathsf{enabled}_{P_1,\sigma}(v_2|v) \implies \neg\mathsf{disabled}_{K_1,ks}(v_2|v)\big) \\ = inv \wedge \big(&\neg w_2 \vee ww_1 \vee u_1 \ \vee \ \neg(l_2^3 \vee l_2^7) \ \vee \ \neg req_1 \vee turn_1 = turn_2\big) \end{aligned}$$

for component P_1/K_1, and similarly for components P_2/K_2 and S/obs. The refinement relation $\leq = \leq_{mutex,ks}^{inv} (\sigma_{P_1}, K_1)$ is shown in Table 4.

In order to prove refinement of P_1 by K_1, we have to show according to Definition 13 that (1) the initial state s_1 of P_1 is refined by some initial state of K_1 (all of whose states are by definition initial states in its trivial unfolding): this condition is obviously satisfied; and (2) for any state q' among the states l_1^1, l_1^2, l_1^5, l_1^6 simulating the initial state s_1 of the projection of σ, we have $\sqsubseteq_{P_1} [(s_1, s_2, idle)/Q_{mutex}, q'/Q_{K_1}] = true$). This condition is satisfied, too. It follows that $K_1 \preceq_{mutex,ks}^{inv} \pi_{P_1}(\sigma)$. Similarly, it can be shown that $K_2 \preceq_{mutex,ks}^{inv} \pi_{P_2}(\sigma)$

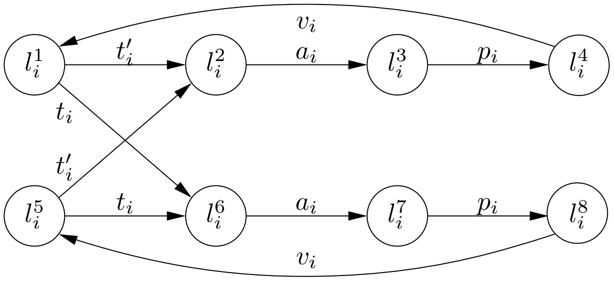


Fig. 8. Behavior of component K_i

Table 2. $\prec$ after 1 iteration and greatest fixpoint $\preceq^{inv}_{mutex,ks}$ (σ_{P_1}, K_1) of $\prec$

	l_1^1	l_1^2	l_1^3	l_1^4	l_1^5	l_1^6	l_1^7	l_1^8
s_1	●	●	-	-	●	●	-	-
w_1	●	●	●	-	●	●	●	-
u_1	-	-	-	●	-	-	-	●

and $obs \preceq^{inv}_{mutex,ks} \pi_S(\sigma)$. The mappings $\sigma_{K_1} := \sigma_{P_1} \searrow_{inv} K_1$, $\sigma_{K_2} := \sigma_{P_2} \searrow_{inv} K_2$, and $\sigma_{obs} := \sigma_S \searrow_{inv} obs$ are equal to the target components K_1, K_2, and *obs*, that is, their full behavior of the target components is used to implement σ. With Theorem 1, global refinement, that is, $ks \preceq^{inv} \sigma$, follows.

In this example, we have mapped σ on a system whose behavior is very close to that of σ. This does not need to be the case, however, as long as the available behavior of the target platform is "expressive enough" to refine σ.

5 Conclusion

We have presented an approach to compositionally cope with strategies, by way of refinement and mapping, in a platform-based design process consisting of successive mapping steps. The latter are performed component-wise; compositionality results ensure that the refinement carries over to the global strategy. The results are formulated in a subset of the heterogeneous component framework BIP. We intend to implement them in the tool Prometheus [9], and apply the approach to models of highly concurrent systems of loosely interacting components, such as sensor networks and genetic networks.

This work opens several interesting research directions. We are currently studying the application of the results presented here to the special case of *reach strategies* which generalize the notion of acyclic paths in closed systems, to strategies ensuring reachability properties.

A rigorous design flow which allows to separately refine components, is crucial to cope with the fast growing complexity of embedded systems. The design flow outlined in Fig. 7 needs to be further developed to be usable in real life:

for instance, it is current practice to refine different components with a common implementation (shared refinement), or to map a component on a set of lower-level components, which is required for co-design. Another direction worth further study is the application to distributed fault-tolerant implementation of component-based systems.

References

1. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Proc. SEFM 2006 (invited paper), pp. 3–12. IEEE, Los Alamitos (2006)
2. Bliudze, S., Sifakis, J.: The algebra of connectors — structuring interaction in BIP. In: Proc. EMSOFT 2007, pp. 11–20. ACM, New York (2007)
3. Broy, M.: Compositional refinement of interactive systems. J. ACM 44(6), 850–891 (1997)
4. Chatterjee, K., Henzinger, T.: Assume-guarantee synthesis. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 261–275. Springer, Heidelberg (2007)
5. de Alfaro, L., Dias da Silva, L., Faella, M., Legay, A., Roy, P., Sorea, M.: Sociable interfaces. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 81–105. Springer, Heidelberg (2005)
6. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proc. 9th Annual Symposium on Foundations of Software Engineering (FSE), pp. 109–120. ACM Press, New York (2001)
7. Fernandez, J.-C., Mounier, L.: "On the Fly" verification of behavioural equivalences and preorders. In: Larsen, K.G., Skou, A. (eds.) CAV 1991. LNCS, vol. 575. Springer, Heidelberg (1992)
8. Gaudin, B., Marchand, H.: An efficient modular method for the control of concurrent discrete event systems: A language-based approach. Discrete Event Dynamic System 17(2), 179–209 (2007)
9. Gössler, G.: Component-based design of heterogeneous reactive systems in prometheus. Research Report 6057, INRIA (2006)
10. Gössler, G., Sifakis, J.: Composition for component-based modeling. Science of Computer Programming 55(1-3), 161–183 (2005)
11. Kessels, J.L.W.: Arbitration without common modifiable variables. Acta Informatica 17(2), 135–141 (1982)
12. Keutzer, K., Malik, S., Newton, A.R., Rabaey, J.M., Sangiovanni-Vincentelli, A.: System level design: Orthogonalization of concerns and platform-based design. IEEE Trans. on Computer-Aided Design 19(12) (2000)
13. Wonham, W.M., Ramadge, P.J.: Modular supervisory control of discrete event systems. Mathematics of Control Signals and Systems 1(1), 13–30 (1988)

A Sound Analysis for Secure Information Flow Using Abstract Memory Graphs

Dorina Ghindici[1], Isabelle Simplot-Ryl[1], and Jean-Marc Talbot[2]

[1] CNRS/INRIA/Univ. Lille 1, France
[2] LIF/CNRS/Univ. de Provence, France

Abstract. In this paper we present a flow-sensitive analysis for secure information flow for Java bytecode. Our approach consists of computing, at all program points, an abstract memory graph (AMG) which tracks how input values of a method may influence its outputs. This computation subsumes a points-to analysis (reflecting how objects depend on each other) by addressing dependencies arising from data of primitive types and from the control flow of the program. Our graph construction is proved to be sound for both intra-procedural and inter-procedural analysis by establishing a non-interference theorem stating that if an output value is unrelated to an input one in the AMG then the output remains unchanged when the input is modified. In contrast with many type-based information flow techniques, our approach does not require security levels to be known during the computation of the graph: security aspects of information flow are checked by labeling ”a posteriori” the AMG with security levels.

1 Introduction

Information flow analysis [18] detects how data may flow between variables focusing on data manipulations of primitive types. This analysis is used to check data propagation in programs with regard to security requirements and aims to avoid that programs leak confidential information: observable/public outputs of a program must not disclose information about secret/confidential values manipulated by the program. *Non-interference* [11] defines the absence of illicit information flows by stating that public outputs of a program remain unchanged if the secret inputs are modified. Data are labeled with security levels, usually $high$ for secret/confidential values and low for observable/public variables. Some information flows arise from assignments (direct flow), others from the control structure of a program (implicit flow). For example, the code `l=h` generates a direct flow from `h` to `l`, while `if(h) then l=1 else l=0` generates an implicit flow still from `h` to `l`. If `h` has security level $high$ and `l` low, then the examples are insecure, as secret data can be infered by the reader of `l`.

In object oriented languages, like Java, the information flow analysis is related to the analysis of references. *Points-to analysis* is a static analysis which computes for every object the set of objects to which it may point to at runtime. Points-to analysis is intensively used in optimizations and as a premiminary part of several static analysis technics such as escape analysis [19,22] or object-oriented program slicing [12].

In this paper we propose a sound flow-sensitive analysis of Java programs that computes an abstract memory graph including references and primitive types. The computed

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 355–370, 2010.

graph is a points-to graph extended with primitive values and dependencies raised by the control flow. This abstract memory graph (denoted AMG) characterizes on one hand how fields of objects point to other objects, and on the other hand the dependencies between primitive values through direct or indirect flow, in the sense of non-interference. This yields an analysis which is more general than “traditional” information flow analysis (in particular, than type-based information flow analysis as initiated in [21]) as it computes an AMG abstracting the dependencies between program data: a node a is related to b if the value of b may influence the value of a. These dependencies are not made explicit in “traditional” information flow analysis and replaced by coarser flows of security levels from an a priori fixed security lattice [7].

In our work, we compute these dependencies between values independently of any *a priori* information like security levels. Therefore, when applied to secure information flow, our approach allows to reuse the same analysis for various security lattices without re-analysing the code.

Let us consider the example in Figure 1, for tax and salary calculations. The AMG of method `tax` contains dependencies between fields `tax` and `sal`, `min`, `max` (due to direct flow) and between `tax` and `avg`, `sal` (due to implicit flow). Considering that the AMG is annotated with security levels $high$ for `sal` field and low for `tax` field, the program is insecure as a $high$ value flows to a low value. The program is secure if we consider a second security policy expressing that both `sal` and `tax` fields are $high$.

The paper is structured as follows: we discuss related work in Section 2. Section 3 presents the JVM model used in this paper and defines non-interference. Section 4 describes the construction of an AMG during the intra-method analysis of sequential programs. We present the correctness of the construction in Section 5, proving a non-interference theorem. Section 6 describes the inter-procedural analysis and adds support for method invocation. Section 7 applies the AMG to information flow.

```
class Rate {
  int avg, min, max;
}

class Income {
  int sal, tax;
  ..
  void tax(Rate p1){
   int tmp;
   if(sal < p1.avg)
     tmp = p1.min;
   else
     tmp = p1.max;
   tax = sal * tmp;}
}
```

Fig. 1. Example

2 Related Work

Information flow enforcement is a well studied area. A considerable amount of work on information flow control, based on static analysis [7,16,21] and on theoretical foundations [1], has been achieved in the last decades. Some of these works have also been impletemented: JFlow [16], for instance, is a powerful tool that permits static type checking of flow annotations in Java programs. In recent years, much of the literature has focused on object-oriented [2,5] languages. In the sequel, we discuss works addressing non-interference for Java(-like) and low-level languages (*eg* Java bytecode).

Most of the approaches for Java are typed-based [5,20]. Their main weakness is that they are flow-insensitive (*i.e.*, they do not take into account the order of statements in a program). The program `l=h; l=0;` (where `h` has the security level $high$ and `l` low) will be rejected, as type systems require every subprogram to be well typed. However, Hunt and Sands proposed recently a family of flow-sensitive type systems [13] for tracking information flow. Typed-based approaches for low-level languages have also been proposed, as in [3,6,15].

Our approach leads to a flow sensitive information flow analysis for object-oriented programs. In this sense, it is similar to [2] where a pointer alias has been used to perform an inter-procedural and flow sensitive information flow analysis for object-oriented programs. However, this work addresses a high level language, including control structures (eg `if-then-else` and `while-do`) and is developed in the style of Hoare logic while we consider a fully static analysis approach.

Program dependence graphs (PDGs) traditionally defined for program slicing can be used for information flow analysis [12]. This approach is indeed more powerful than type based techniques as PDG's are flow sensitive. But this latter approach differ from ours: AMG, the graph built by our method and PDG's don't have the same kind. Though PDG's may contain nodes representing objects (for handling alias information), most of their nodes represent statements of the program (one node per statement), whereas nodes in AMG represent objects and values manipulated by the program. Hence, due to the use of allocation site model in our method, an AMG contains roughly one node per bytecode like `new`, `iconst_`n making it much smaller than a PDG.

As in [8], we address low-level programs with a context- and flow-sensitive approach based on abstract interpretation for mono-threaded bytecode; however, in [8], the abstract domain is restricted to boolean functions leading to a field- and object insentivite analysis whereas we rely on an accurate points-to analysis to model in a precise way object references and values of object fields.

Most of the previous cited works require the lattice security model of information flow to be known from the very beginning of the analysis. We believe that *traditional* information flow analysis can be recovered as an abstract interpretation of our AMGs, this abstract interpretation taking into account the security levels.

3 Non-interference for Java Virtual Machine Programs

3.1 Notation

We consider finite directed graphs whose edges are labelled by elements of the set $\mathcal{L}$: a graph G is given by (V, E) where V is its set of vertices (or nodes), and $E \subseteq V \times V \times \mathcal{L}$ is its set of labelled edges. In $G = (V, E)$, the edge from vertex u to v, labeled with l is denoted (u, v, l), $adj_G(u, l)$ is the set of adjacent vertices of u in G, reached by an edge labeled by l, and a node u is a leaf if for any node v and label l, $(u, v, l) \notin E$.

A vertex v is reachable from u in a graph G if there is a path (a sequence of edges leading) from u to v and we denote by $Reach_G(u)$ the set of vertices reachable from u in G. We define $G\lfloor u \rfloor$ the subgraph of $G = (V, E)$ given by $(Reach_G(u), \{(v, w, l) \mid (v, w, l) \in E \text{ and } v \in Reach_G(u)\})$. The union of two graphs $(V_1, E_1) \cup (V_2, E_2)$ is the graph $(V_1 \cup V_2, E_1 \cup E_2)$; the graph inclusion ($G_1 \subseteq G_2$) is defined accordingly.

For a graph $G = (V, E)$, $G[(u, l) \mapsto v]$ agrees with G except that all the edges of the form (u, u', l) in E are replaced by a unique edge (u, v, l). Finally, for a function f, by $f[x \mapsto e]$ we denote the function f' such that $f'(y) = f(y)$ if $y \neq x$ and $f'(x) = e$.

3.2 The Java Virtual Machine Model

We consider a set of class names $\mathit{Classes}$, a set of methods $\mathit{Methods}$ and a set Fields of field names. The set of JVM values is defined as $Jv = \mathit{Val} \cup \mathit{Obj}$ where Val is the set of primitive values, and Obj the set of objects, including the special value `null`. The function $\mathit{Type} : \mathit{Obj} \rightarrow \mathit{Classes}$ returns the type of an object.

Due to limited space, we focus on the formal model of a subset of the JVM instruction set: `prim` *op* stands for a primitive operation taking two operands, pushing the result on the stack; `pop` pops the top of the stack. `iconst_`n (resp. `aconst_null`) pushes the primitive value n (resp. null) on the stack. `new` C creates new object of type C in the memory. `goto` a jumps to address a and `ifcmp` a does the same only if the top of the stack is greater or equal to second element on the stack. `load` x pushes the content of the local variable x on the stack whereas `store` x pops the top of the stack and stores it into the local variable x. `getfield` $f_{C'}$ loads on the stack the field $f_{C'}$ of the object at the top of the stack and `putfield` $f_{C'}$ stores the top of the stack in the field $f_{C'}$ of an object on the stack. Finally, `invoke` $m_{C'}$ is the virtual invocation of method $m_{C'}$ and `areturn` returns an object and exit the method.

Support for full Java is presented in [10] and [9]. We assume the bytecode programs to be well-typed, to successfully pass the class file verifier and we deal only with executions that terminate and do not throw exceptions.

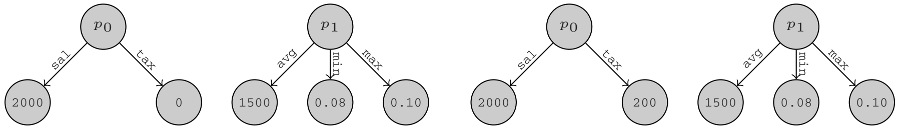
Memory model. We represent the memory heap as a directed graph[1], *i.e.* a *memory graph*, G which is a triple (V, E, ς). A node of V designates a location: V is partitioned into $\mathcal{O}(V) \subseteq \mathit{Obj}$ for locations containing object references and $\mathcal{V}(V)$ for locations containing values of Val. For short, we use $\mathcal{O}(G)$ and $\mathcal{V}(G)$ instead of $\mathcal{O}(V)$ and $\mathcal{V}(V)$ respectively. Edges from E represent field references and are labeled with field names from Fields. As values contained in locations may change during the execution, we use an injective function $\varsigma : \mathcal{V}(V) \rightarrow \mathit{Val}$ that labels each node from $\mathcal{V}(V)$ with the value stored in this location; the (unique) node `null` as well as nodes of $\mathcal{V}(V)$ are exactly the leaves of memory graphs. For each primitive field of an object, the memory has a location (node) containing (labeled with) the value of the field and this vertex is the target of a unique edge.

We consider an allocator function $G' = \mathit{new}(o, C)$ which creates a new graph structure for an object named o of type C; o is the root of the graph. The graph G' contains the initial values of the new created object (vertices containing 0 for fields of primitive type and edges to `null` for fields of type object).

For a memory graph $G = (V, E, \varsigma)$, $G[u \mapsto l]$ designates the graph $(V, E, \varsigma[u \mapsto l])$. We naturally extend the definition of union, inclusion, $G\lfloor u \rfloor$, and $G[(u, f) \mapsto v]$ to memory graphs. Moreover, we write $G_1 \equiv G_2$ when the memory graphs G_1 and G_2 are isomorphic (considering both edge and vertex labelling).

[1] The advantage of this representation is the independence between objects and actual locations where these objects are allocated. However, the graph representing the memory is isomorphic to any address assignment in an execution. This independence property is crucial when comparing memories for two executions of the same method for different input values.

Operational semantics. For a method m of class C, n_m denotes the number of its arguments, χ_m its set of local variables, P_m its instruction list, $P_m[i]$ the ith bytecode of the method m. A (concrete) state is a pair $Q = (\mathit{fr}, G)$, where fr represents a stack of frames corresponding to a dynamic chain of method calls (with the current frame on the top), and G is a memory graph. A frame is a triple (pc, ρ, s), where $pc < |P_m|$ is the program counter (*i.e.*, an index into P_m), $\rho : \chi_m \to Jv$ represents a value assignment of local variables and s is a stack with elements in Jv. The operational semantics of Java bytecode is presented in Figure 3. The concatenation $n :: s$ denotes a stack having n on top, while $f_{C'}$ designates the field f of the class C'.



(a) After `0` (b) After `17`

Fig. 2. Memory Graph for `tax` method

Figure 2 shows memory graphs of the `tax` method (Figure 2(a) at the beginning of the method, Figure 2(b) after executing the entire method). For instance, the `putfield` rule depends on the type of the manipulated field: changing the value of a primitive field means changing the label of a vertex (*e.g.* instruction 17 actually modifies the label of the memory location corresponding to the `tax` field), whereas changing an object field consists of changing the edge to the vertex containing the new pointed object.

The intra-method control flow graph CF_m of a method m is the graph with P_m as set of vertices and edges from i to each elements of $succ(i)$, with $succ(i)$ being (for $i \in P_m$): $\{a\}$ if $P_m[i] = \texttt{goto}\ a$, $\{a, i+1\}$ if $P_m[i] = \texttt{ifcmp}\ a$, $\varnothing$ if $P_m[i] = \texttt{areturn}$ and $\{i+1\}$ otherwise. We also use the notation $pred(i) = \{j \mid i \in succ(j)\}$. An exit-point in a control flow graph is a node without successor.

For a method m, a *block* B is a subset of the instruction list P_m together with a distinguished instruction $\mathsf{Entry}(B)$, called the *entry-point* of B such that the control flow graph CF_B of B is a subgraph of CF_m; all vertices in CF_B are reachable from $\mathsf{Entry}(B)$ and CF_B has a unique exit-point $\mathsf{Exit}(B)$. We assume that P_m is itself a block by adding, if needed, a unique (fake) exit-point to its control flow graph. A state $Q = ((i, \rho, s) :: \mathit{fr}, G)$ is a *good initial* state for a block B, if $B[i]$ is defined and executing the block B starting from the state Q terminates in a state denoted by $instr_B(Q)$.

We use the notion of postdominance of [4]: let B be a block. Then, in CF_B, its control flow graph, a node n' post-dominates a node n if $n' \neq n$ and n' belongs to every path from n to $\mathsf{Exit}(B)$. We denote $PD(n)$ the set of post-dominators of n. The immediate post-dominator of n, $ipd(n) \in PD(n)$ satisfies that $\forall n' \in PD(n)$, if $n' \neq ipd(n)$, then $n' \in PD(ipd(n))$. In B, the *dependency region* of a conditional instruction $B[i] = \texttt{ifcmp}\ a$ is the set of instructions executed under this condition: $reg(i) = Reach_{\mathit{CF}_B}(i) \smallsetminus Reach_{\mathit{CF}_B}(ipd(i))$. We define a function $cxt : B \to \wp(B)$ representing the context (the set of conditional bytecodes) under which each instruction is executed: $cxt(i) = \{j \mid i \neq j \wedge i \in reg(j)\}$.

$$\frac{P_m[i] = \mathtt{prim}\,op \qquad n = op(n_1, n_2)}{F' \longrightarrow ((i+1, \rho, n :: s) :: fr, G)} \qquad \frac{P_m[i] = \mathtt{goto}\ \ a}{F \longrightarrow ((a, \rho, s) :: fr, G)}$$

$$\frac{P_m[i] = \mathtt{aconst\_null}}{F \longrightarrow ((i+1, \rho, \mathtt{null} :: s) :: fr, G)} \qquad \frac{P_m[i] = \mathtt{pop}}{((i, \rho, n :: s) :: fr, G) \longrightarrow ((i+1, \rho, s) :: fr, G)}$$

$$\frac{P_m[i] = \mathtt{iconst\_}n}{F \longrightarrow ((i+1, \rho, n :: s) :: fr, G)} \qquad \frac{P_m[i] = \mathtt{new}\ \ C \qquad o = C_i^{\mathsf{fresh}(C_i, G)} \qquad G' = new(o, C)}{F \longrightarrow ((i+1, \rho, o :: s) :: fr, G \cup G')}$$

$$\frac{P_m[i] = \mathtt{ifcmp}\ \ a \qquad n_1 < n_2}{F' \longrightarrow ((i+1, \rho, s) :: fr, G)} \qquad \frac{P_m[i] = \mathtt{ifcmp}\ \ a \qquad n_1 \geq n_2}{F' \longrightarrow ((a, \rho, s) :: fr, G)}$$

$$\frac{P_m[i] = \mathtt{load}\ \ x}{F \longrightarrow ((i+1, \rho, \rho(x) :: s) :: fr, G)} \quad \frac{P_m[i] = \mathtt{store}\ \ x}{((i, \rho, n :: s) :: fr, G) \longrightarrow ((i+1, \rho[x \mapsto n], s) :: fr, G)}$$

$$\frac{P_m[i] = \mathtt{getfield}\ \ f_{C'} \qquad n \neq \mathtt{null} \qquad n' \in adj_G(n, f_{C'}) \qquad n' \in Obj}{((i, \rho, n :: s) :: fr, G) \longrightarrow ((i+1, \rho, n' :: s) :: fr, G)}$$

$$\frac{P_m[i] = \mathtt{getfield}\ \ f_{C'} \qquad n \neq \mathtt{null} \qquad n' \in adj_G(n, f_{C'}) \qquad n' \notin Obj}{((i, \rho, n :: s) :: fr, (V, E, \varsigma)) \longrightarrow ((i+1, \rho, \varsigma(n') :: s) :: fr, (V, E, \varsigma))}$$

$$\frac{P_m[i] = \mathtt{putfield}\ \ f_{C'} \qquad n \neq \mathtt{null} \qquad v \in Val \qquad n' \in adj_G(n, f_{C'})}{((i, \rho, v :: n :: s) :: fr, G) \longrightarrow ((i+1, \rho, s) :: fr, G[n' \mapsto v])}$$

$$\frac{P_m[i] = \mathtt{putfield}\ \ f_{C'} \qquad n \neq \mathtt{null} \qquad v \notin Val}{((i, \rho, v :: n :: s) :: fr, G) \longrightarrow ((i+1, \rho, s) :: fr, G[(n, f_{C'}) \mapsto v])}$$

$$\frac{P_m[i] = \mathtt{invoke}\ \ m_{C'} \qquad o \neq \mathtt{null}}{((i, \rho, p_{n_m} :: \cdots :: p_1 :: o :: s) :: fr, G) \rightarrow ((0, \{0 \mapsto o, 1 \mapsto p_1 \ldots n_m \mapsto p_{n_m}\}, \epsilon) :: (i, \rho, s) :: fr, G)}$$

$$\frac{P_m[i] = \mathtt{areturn}}{((i, \rho, v :: s) :: (i', \rho', s') :: fr, G) \longrightarrow ((i'+1, \rho', v :: s') :: fr, G)}$$

where $F = ((i, \rho, s) :: fr, G)$, $F' = ((i, \rho, n_1 :: n_2 :: s) :: fr, G)$. $\mathtt{prim}\ op$ stands for primitive operations with two parameters. The function $\mathsf{fresh}(c, G)$ returns the least natural k such that c^k is not used as a vertex label in G.

Fig. 3. A subset of operational semantics rules

3.3 Non-interference

Our analysis computes an abstraction of the links between values of different objects. We formalize this notion of dependency through non-interference: roughly, at the method level, the non-interference between an object o and an input value ι (of primitive type) can be stated as "changing of value of ι does not affect the value of o". The "value" of an object is defined by the values of its fields of primitive type and recursively, by the "values" of its object fields. An input value of a block is a value of primitive type chosen in the concrete state before the execution of the block; formally:

Definition 1 (Set of input values). *Let $Q = ((i, \rho, s) :: fr, G)$ be a state. Then, the set of* input values *of Q is $\mathcal{I}(Q) = \mathcal{V}(G) \cup \{x \in \chi_m \mid \rho(x) \in Val\}$.*

Definition 2 (Value-change). *A* value-change *of a state $Q = ((i, \rho, s) :: fr, G)$ and an input value ι from $\mathcal{I}(Q)$ is a state $Q' = ((i, \rho', s) :: fr, G')$ such that for some value a of primitive type, either $\iota \in \mathcal{V}(G)$, $\rho = \rho'$ and $G' = G[\iota \mapsto a]$, or $\iota \in \{x \in \chi_m \mid \rho(x) \in Val\}$, $\rho' = \rho[\iota \mapsto a]$ and $G' = G$.*

Definition 3 (Non-interference). *For any block B and any state Q, let G, G_B be the memory graphs of Q and $instr_B(Q)$ respectively. For any input value ι from $\mathcal{I}(Q)$ and for any object node o in G, ι* does not interfer *in B* with *o if for any value-change Q' of Q and ι, one has $G_B\lfloor o \rfloor \equiv G'_B\lfloor o \rfloor$, G'_B being the memory graph of $instr_B(Q')$.*

4 Intra-Method Abstract Dependency

The core of our analysis is an algorithm that builds an AMG, that is an "abstraction" of the memory graph containing all possible dependencies between objects and input values. In this section, we present the intra-method analysis focusing on programs without method calls: the algorithm consists in computing an AMG for each program point of a method. Inter-method analysis will be addressed in Section 6.

```
0: load p0
1: getfield sal
2: load p1
3: getfield avg
4: ifcmp 9
5: load p1
6: getfield min
7: store tmp
8: goto 12
9: load p1
10: getfield max
11: store tmp
12: load p0
13: load p0
14: getfield sal
15: load tmp
16: imul
17: putfield tax
18: return
```

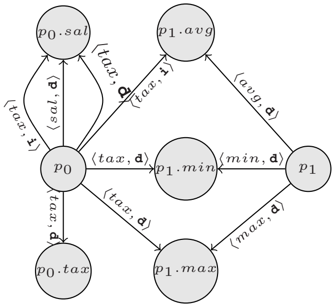

$p_0.sal$
$p_1.avg$
p_0
$p_1.min$
p_1
$p_0.tax$
$p_1.max$
$\langle tax, \mathbf{i}\rangle$
$\langle sal, \mathbf{d}\rangle$
$\langle tax, \mathbf{d}\rangle$
$\langle tax, \mathbf{i}\rangle$
$\langle avg, \mathbf{d}\rangle$
$\langle tax, \mathbf{d}\rangle$
$\langle min, \mathbf{d}\rangle$
$\langle tax, \mathbf{d}\rangle$
$\langle max, \mathbf{d}\rangle$
$\langle tax, \mathbf{d}\rangle$

Fig. 4. Example

The abstract model: AMGs Let us present the abstract model on the example in Figure 1. The Java bytecode of method `tax` and the AMG obtained after the analysis of the method are depicted in Figure 4.

The AMG G at a program point is a representation of the memory such that, when restricted to objects, G and the memory graph are related by an abstraction relation. Then this abstraction is extended with primitive values and implicit flow dependencies. Nodes in AMG contain abstractions of JVM objects, constants and newly created objects as well as initial values of primitive type of the method. For an AMG $G = (V, E)$, $\mathcal{O}(V) = \mathcal{O}(G)$ denotes the set of nodes abstracting JVM objects and $\mathcal{V}(V) = \mathcal{V}(G)$, nodes abstracting primitive nodes.

Hence, nodes of the graph G are defined to take into account the model:

- $n^n_{pc}, pc < |P_m|$: node modeling all the objects created by the execution of the object allocation instruction pc [2]. As for memory graphs, the graph structure rooted at n^n_{pc} contains nodes for each primitive field, and edges to the node $n^{\texttt{null}}_{-1}$, which is the abstraction of `null`, for object fields,
- $n^c_{pc}, pc < |P_m|$: constant value "created" at instruction pc. There is a unique node for every constant creation statement in the method.

[2] We use the *object allocation site model*: all objects created at the same program statement have the same abstraction.

Edges represent flows (*i.e.*, dependencies) between nodes; they are labeled with pairs $\langle f, t\rangle$, where $f \in \mathit{Fields}$ is the name of the field and $t \in \mathcal{F} = \{\mathtt{d}, \mathtt{i}\}$ is the type of flow: $\mathtt{i}$ for implicit flow and $\mathtt{d}$ for direct flows, with the order relation $\mathtt{i} \sqsubseteq \mathtt{d}$. In Figure 4, the edge labeled $\langle tax, \mathtt{d}\rangle$ between p_0 and $p_0.sal$ means that there is direct flow, arising from an assignment, from $p_0.sal$ to the field tax of p_0. The edge labeled $\langle tax, \mathtt{i}\rangle$ between p_0 and $p_0.sal$ shows that the value of $p_0.sal$ might be deduced from the field tax of p_0, as it depends on the condition tested at instruction 4.

To deal with implicit flows, we use the notion of regions. When executing an instruction i which adds an edge $(u, v, \langle f, \mathtt{d}\rangle)$, edges having the form $(u, v', \langle f, \mathtt{i}\rangle)$ are also added, where v' is the value tested by conditional instructions in $cxt(i)$. The intuition behind is that someone who knows the control structure of the program and can observe the field f of u, is able to deduce the value of v'.

Implicit flows are from objects modified inside a region to values on which the execution of the region depends. These facts allow us to deduce some properties of an AMG G: *(i)* any node $u \in \mathcal{V}(G) \cup \{n^{\mathtt{null}}_{-1}\}$ is a leaf, and *(ii)* for any edge of type $(u, u', \langle f, \mathtt{i}\rangle) \in E$, $u \in \mathcal{O}(G) \setminus \{n^{\mathtt{null}}_{-1}\}$ and $u' \in \mathcal{V}(V)$. These properties state that edges between two references are always direct edges: if $u_1, u_2 \in \mathcal{O}(G)$ and $(u_1, u_2, \langle f, t\rangle) \in E$ then $t = \mathtt{d}$. Value nodes are always leaves, and implicit edges are always between a reference and a value node. Thus, edges to primitive values are not imprecated in the graph restricted to objects. Our construction is a points-to graph prolonged by edges about primitive values and implicit flows.

Abstract semantics. Building the AMG requires to model local variables and stack contents, and to deal with implicit flow, we must know the conditions under which the local variables and the stack are modified. Thus, elements from stack and local variables have the form (u, t) where $u \in V$ and $t \in \mathcal{F}$. Hence, an abstract state is of the form $Q = (\rho, s, (V, E))$ where (V, E) is the AMG, ρ is a mapping from χ_m to $\wp(V \times \mathcal{F})$ and s is the stack with elements in $\wp(V \times \mathcal{F})$.

For each bytecode b, we define an abstract rule $Q' = \overline{instr}_b(Q, \Gamma)$ where Γ is the set of nodes u corresponding to values on which depends the execution of the bytecode, reflecting the impact of the implicit flow. The abstract rules are presented in Figure 5.

To reflect the impact of control regions on stack and local variables, every instruction modifying these latter takes into consideration the values in the context. For example, the instruction `new` pushes on the stack a new abstract value as well as the context under which the operation takes place, that is pairs from TV_Γ. The `store` bytecode not only stores the top of the stack in the local variable array, but also TV_Γ.

The instruction `getfield` pushes on the stack the adjacent of u, if u is an object reference, or keeps u on the stack if it is a primitive value arising from implicit flow.

The most significant bytecode is `putfield`, as it modifies the AMG. Apart edges from direct nodes in u (having the form $(e, \mathtt{d})$) to elements in v, implicit flow edges from the nodes e ($(e, \mathtt{d}) \in u$) to nodes in Γ, the instruction adds implicit flow edges from nodes e ($(e, \mathtt{d}) \in u$) to implicit nodes e' ($(e', \mathtt{i}) \in u$). The presence of nodes like $(e', \mathtt{i})$ in u means that the objects in u depend on e'. Thus, we propagate the implicit dependencies of an object to every field being modified of that object.

For instance, the `putfield` bytecode at instruction 17 in Figure 4 assigns the field tax of p_0 and generates the following edges labeled: (i) $\langle tax, \mathtt{d}\rangle$ from p_0 to $p_0.sal$,

$$\frac{(\rho, s, G)}{(\rho, s, G)}\,\texttt{goto}\ a \qquad \frac{(\rho, v_1 :: v_2 :: s, G)}{(\rho, v_1 \cup v_2 \cup \mathit{TV}_\Gamma :: s, G)}\,\texttt{prim}\ op \qquad \frac{(\rho, s, G)}{(\rho, \{(n_i^n, \mathtt{d})\} \cup \mathit{TV}_\Gamma :: s, G)}\,\texttt{new\_i}\ C$$

$$\frac{(\rho, s, G)}{(\rho, \{(n_{-1}^{\text{null}}, \mathtt{d})\} \cup \mathit{TV}_\Gamma :: s, G)}\,\texttt{aconst\_null} \qquad \frac{(\rho, v :: s, G)}{(\rho, s, G)}\,\texttt{pop} \qquad \frac{(\rho, s, G)}{(\rho, \{(n_i^c, \mathtt{d})\} \cup \mathit{TV}_\Gamma :: s, G)}\,\texttt{iconst\_i}v$$

$$\frac{(\rho, v_1 :: v_2 :: s, G)}{(\rho, s, G)}\,\texttt{ifcmp}\ a \qquad \frac{(\rho, s, G)}{(\rho, \rho(x) \cup \mathit{TV}_\Gamma :: s, G)}\,\texttt{load}\ x \qquad \frac{(\rho, u :: s, G)}{(\rho[x \mapsto u \cup \mathit{TV}_\Gamma]s, G)}\,\texttt{store}\ x$$

$$\frac{(\rho, u :: s, G)}{(\rho, \{(e, t) \mid e \in adj_G(e', \langle f_{C'}, t\rangle) \wedge e' \in V_{\mathtt{d}}^u\} \cup \{(e, \mathtt{i}) \mid e \in V_{\mathtt{i}}^u\} \cup \mathit{TV}_\Gamma :: s, G)}\,\texttt{getfield}\ f_{C'}$$

$$\frac{(\rho, v :: u :: s, (V, E))}{\left(\rho, s, \left(V, \begin{array}{l} E\ \cup\{(e, e', \langle f_{C'}, t\rangle) | (e, \mathtt{d}) \in u, e \neq n_{-1}^{\text{null}}, (e', t) \in v\} \\ \quad\ \cup\{(e, e', \langle f_{C'}, \mathtt{i}\rangle) | (e, \mathtt{d}) \in u, e \neq n_{-1}^{\text{null}}, e' \in \Gamma \cup V_{\mathtt{i}}^u\} \end{array}\right)\right)}\,\texttt{putfield}\ f_{C'}$$

with $V_{\mathtt{d}}^u = \{e \mid (e, \mathtt{d}) \in u\}$ $V_{\mathtt{i}}^u = \{e \mid (e, \mathtt{i}) \in u\}$ $\mathit{TV}_\Gamma = \{(e, \mathtt{i}) \mid e \in \Gamma\}$

Fig. 5. Subset of the abstract transformation rules

$p_1.min$, $p_1.max$ and (ii) $\langle tax, \mathtt{i}\rangle$ from p_0 to tmp, was modified in a control region depending on these two fields.

Algorithm. Our analysis is a flow sensitive may-analysis [19], computing an AMG at each program point as the union of graphs created by all the execution paths reaching that point; it is defined in the context of a monotone framework [17] for data flow analysis. To comply with this framework, we define an order relation $\sqsubseteq$ and a join operator $\sqcup$ on the property space $\mathcal{S}$ (the set of pairs (Q, Γ) augmented with $\perp$ the neutral element of $\sqcup$) such that $(\mathcal{S}, \sqsubseteq, \sqcup)$ forms a semi-joint lattice which satisfies the Ascending Chain Property (all increasing sequences in $\mathcal{S}$ become eventually constant).

The execution of a method is abstracted as a set of equations based on the abstract rules $\overline{instr}_b$: the analysis of a block of instructions $B \subseteq P_m$ of a method m, assumed to be represented by its control flow graph CF_B, is described by an equation system $\mathcal{E}_B$, starting from a given initial state (Q_0, Γ_0) (recall that we consider terminating executions without exceptions and we consider here initial states that allow such executions). For every node i in CF_B, (Q_i, Γ_i) represents the state and the context (required by the implicit flow) under which the instruction i is executed. The context represents the conditions tested (the top of the stack) by instructions in $cxt(i)$. Thus, for all i in CF_B:

$$\begin{aligned}(Q_i, \Gamma_i) = (S_Q \sqcup \textstyle\bigsqcup_{j \in pred(i)} \overline{instr}_{B[j]}(Q_j, \Gamma_j),& \\ S_\Gamma \cup \{u \mid (u, t) \in v \text{ with } Q_k = (\rho, v :: s, G), k \in cxt(i)\})&\end{aligned} \tag{1}$$

where (S_Q, S_Γ) equals to (Q_0, Γ_0) if $i = \mathsf{Entry}(B)$ and to $(\perp, \varnothing)$ otherwise.

The abstract rules are monotone with respect to the ordering relation $\sqsubseteq$, thus we can solve the system (1) using standard methods for monotone dataflow analysis. For a block B and a pair (Q, Γ), we define $\overline{instr}_B(Q, \Gamma)$ as the state $\overline{instr}_{B[e]}(Q_e, \Gamma_e)$

where $e = \mathsf{Exit}(B)$ and $(Q_e, \mathit{\Gamma}_e)$ is obtained in the least solution of the system of equations $\mathcal{E}_B$ starting from the initial state $(Q, \mathit{\Gamma})$.

5 Soundness of the Intra-method Analysis

We now prove the correctness of the AMG construction with respect to non-interference: if there is no dependency (path in the abstract graph) between an object and an input value, then changing the input value in the concrete graph will not affect the concrete graph of the object. As we compare, in this section, concrete and abstract worlds, we denote by e a concrete element and by $\overline{e}$ an abstract element. Note that $\overline{e}$ is just an abstract element, and not necessarily the abstraction of e.

5.1 Relations between Abstract and Concrete Worlds

We first formally relate abstract and concrete semantics: we consider an abstraction function to relate memory graphs restricted to objects nodes and AMGs. Because of the allocation site model, it is possible to relate concrete nodes to abstract nodes in a non ambiguous manner. However, for complete graphs (including nodes representing values), we cannot define an abstract relation as it is not possible to associate uniquely a concrete node value with an abstract one in the model we use.

Definition 4 **(α-abstraction).** *Let G be a memory graph and $\overline{G}$ be an AMG. An* abstraction function *α from G to $\overline{G}$ is a mapping from $\mathcal{O}(G)$ to $\mathcal{O}(\overline{G})$ respecting the allocation site model (for all k, all the C_i^k are mapped to n_i^n, and $\alpha(\texttt{null}) = n_{-1}^{\texttt{null}}$) and the graph $\alpha(G) = (\{\alpha(u) \mid u \in \mathcal{O}(G)\}, \{(\alpha(u), \alpha(v), \langle e, \boldsymbol{d}\rangle) \mid u, v \in \mathcal{O}(G) \wedge (u, v, e) \in E\})$ is included into $\overline{G}$.*

We say then that $\overline{G}$ is an α-abstraction of G, denoted by $G \lhd^{\alpha} \overline{G}$.

The abstraction function is carried over concrete and abstract states.

Definition 5 **(State abstraction).** *Let $Q = ((pc, \rho, s) :: fr, G)$ be a concrete state, $\overline{Q} = (\overline{\rho}, \overline{s}, \overline{G})$ be an abstract state and α be an abstraction function from $\mathcal{O}(G)$ to $\mathcal{O}(\overline{G})$. $\overline{Q}$ is an α-abstraction of Q (denoted $Q \lhd^{\alpha} \overline{Q}$) if $G \lhd^{\alpha} \overline{G}$, $s \lhd^{\alpha} \overline{s}$, and $\rho \lhd^{\alpha} \overline{\rho}$, with: $s \lhd^{\alpha} \overline{s}$ if $\overline{s}$, s are both empty or if $s =v :: s_1$, $\overline{s} =\overline{v} :: \overline{s_1}$, $s_1 \lhd^{\alpha} \overline{s_1}$, and $(\alpha(v), \boldsymbol{d}) \in \overline{v}$ if $v \in \mathcal{O}(G)$, the local variables abstraction $\rho \lhd^{\alpha} \overline{\rho}$ is defined similarly.*

We now extend the abstraction relation to take into account nodes of primitive type that correspond to input values.

Definition 6 **(α-abstraction extension).** *Let $Q = ((i, \rho, s) :: fr, (V, E, \varsigma))$ be a concrete state, and $\overline{Q} = (\overline{\rho}, \overline{s}, (\overline{V}, \overline{E}))$ such that $Q \lhd^{\alpha} \overline{Q}$ for an abstraction function α. Then, $\alpha_Q^{\overline{Q}} : \mathcal{I}(Q) \longrightarrow \wp(\mathcal{V}(\overline{V}))$ is the unique extension of α in Q if*

$\forall \iota \in \mathcal{I}(Q) \cap \mathcal{V}(V)$, $\alpha_Q^{\overline{Q}}(\iota) = \{\overline{\iota} \mid \exists(\alpha(u), \overline{\iota}, \langle f, \boldsymbol{d}\rangle) \in \overline{E}$, *with* $(u, \iota, f) \in E\}$

$\forall \iota \in \mathcal{I}(Q) \cap \chi_m, \quad \alpha_Q^{\overline{Q}}(\iota) = \{\overline{e} \mid (\overline{e}, \boldsymbol{d}) \in \overline{\rho}(\iota))\}$

Definition 7. *Let $\overline{Q} = (\overline{\rho}, \overline{s}, (\overline{V}, \overline{E}))$ be an abstract state. Let $\beta \subseteq \mathcal{V}(\overline{V})$. For any $\overline{o} \in \mathcal{O}(\overline{V})$, we have free$(\overline{o}, \beta, \overline{Q})$ if one of the following conditions holds:*

- *$\exists(\overline{u}, \overline{o}, \langle f, \mathbf{d} \rangle) \in \overline{E}$ and $\beta \cap adj_{(\overline{V}, \overline{E})}(\overline{u}, \langle f, \mathbf{i} \rangle) = \varnothing$,*
- *$\exists x$ such that $(\overline{o}, \mathbf{d}) \in \overline{\rho}(x)$ and $\nexists(\overline{\iota}, \mathbf{i}) \in \overline{\rho}(x)$ with $\overline{\iota} \in \beta$,*
- *$\exists i$ such that $(\overline{o}, \mathbf{d}) \in \overline{s}[i]$ and $\nexists(\overline{\iota}, \mathbf{i}) \in \overline{s}[i]$ with $\overline{\iota} \in \beta$.*

The function *free*$(\overline{o}, \beta, \overline{Q})$ expresses that $\overline{o}$ is not bound to any $\overline{\iota}$ of β in $\overline{Q}$, meaning that at least one occurrence of $\overline{o}$ on stack, local variables or in AMG does not implicitly depend on β. As varying ι may influence the execution path, and thus the objects being created, this condition ensures, in the following non-interference theorem, the existence of the object o (with $\alpha(o) = \overline{o}$) after any value-change from β.

Theorem 1 (Non-interference). *Let B be an instruction block, Q be a concrete state with G as memory graph, $\overline{Q}$ be an abstract state and α be an abstraction function such that $Q \lhd^{\alpha} \overline{Q}$. Let $\overline{G'}$ be the abstract graph of $\overline{instr}_B(\overline{Q}, \Gamma)$. For an object node o in $\mathcal{O}(G)$ and an input value ι from $\mathcal{I}(Q)$ such that free$(\alpha(o), \alpha_Q^{\overline{Q}}(\iota), \overline{Q})$, if Reach$_{\overline{G'}}(\alpha(o) \cap \alpha_Q^{\overline{Q}}(\iota)) = \varnothing$ then ι does not interfere with o in B.*

5.2 Analysis Correctness

We show first that when the AMGs are restricted to references, our analysis is a sound points-to analysis. We rely on the dataflow framework, slightly adapted to take into account the flow information of the Γ's.

Proposition 1 (Points-to correctness). *Let B be a block, Q be a good initial (concrete) state for B, α be an abstraction function, and $\overline{Q}$ be an abstract state. Then, for any set of abstract values Γ, $Q \lhd^{\alpha} \overline{Q}$ implies $instr_B(Q) \lhd^{\alpha} \overline{instr}_B(\overline{Q}, \Gamma)$.*

The complete proof is presented in [10]. We now prove the correctness of the primitive edges (implicit and direct flow) as a non-interference theorem, relying on the correctness of the points-to analysis.

To prove the non-interference theorem, according to Definition 3, we need to make values vary at some program point according to Definition 2 and to check the impact of this variation on objects at another program point. Thus, we first define the notion of *state variation* that captures how a concrete execution state, corresponding to a program point i, might change when an input value ι has changed in the past of this execution. Definition 8 contains an over estimation of the set of states that the JVM can reach after this change: the main impact is on the memory graph, but the local variables and stack can also be affected. The correctness of the definition is proved later in Proposition 2.

Definition 8 (State variation). *Let $Q = ((i, \rho, s) :: fr, G)$ and $Q' = ((i, \rho', s') :: fr, G')$ be two concrete states. Let $\overline{Q} = (\overline{\rho}, \overline{s}, \overline{G})$ be an abstract state. Let α be an abstraction function. Let $\beta \subseteq \mathcal{V}(\overline{G})$ be a set of nodes of primitive type.*

Then Q and Q' are state variation *from each other with respect to $\overline{Q}$ and β, denoted by $Q \overset{\overline{Q}, \beta}{\longleftrightarrow} Q'$, if $Q \lhd^{\alpha} \overline{Q}$, $Q' \lhd^{\alpha} \overline{Q}$ and*

- *$\forall x, \rho(x) = \rho'(x) \vee \exists \overline{\iota} \in \beta$ such that $(\overline{\iota}, t) \in \overline{\rho}(x)$,*

- $\forall i, s[i] = s'[i] \vee \exists \overline{\iota} \in \beta$ *such that* $(\overline{\iota}, t) \in \overline{s}[i]$,
- *for all* $v \in \mathcal{O}(G) \cup \mathcal{O}(G')$, *either* $v \in \mathcal{O}(G) \cap \mathcal{O}(G')$ *or* $\neg \mathit{free}(\alpha(v), \beta, \overline{Q})$,
- *for all* v *in* $\mathcal{O}(G) \cap \mathcal{O}(G')$, *for all fields* f *from* $\mathit{Type}(v)$,
  - *either there exists a unique node* w *such that* (v, w, f) *is an edge from* G *and from* G' *and, moreover if* $w \in \mathcal{V}(G')$, *then* w *is labeled similarly in* G *and* G'.
  - *or some edge* $(\alpha(v), \overline{\iota}, \langle f, t\rangle)$ *exists in* $\overline{G}$ *for some* $\overline{\iota}$ *in* β.

From state variation definition, we can state that a variation of a state regarding to a set β, does not impact the objects o which have no dependence to any node of β.

Lemma 1. *Let* α *be an abstraction function,* $Q = ((i, \rho, s) :: fr, G)$, *and* $\overline{Q} = (\overline{\rho}, \overline{s}, \overline{G})$ *with* $Q \lhd^{\alpha} \overline{Q}$. *Let* $\beta \subseteq \mathcal{V}(\overline{G})$, *and* $o \in \mathcal{O}(G)$ *such that there exists no path from* $\alpha(o)$ *to any node of* β *in* $\overline{G}$. *Then, in any state variation* $Q' \overset{\overline{Q},\beta}{\longleftrightarrow} Q$ *with* $Q' = ((i, \rho', s') :: fr, G')$, *if* $\mathit{free}(\alpha(o), \beta, \overline{Q})$, *we have* $G\lfloor o \rfloor \equiv G'\lfloor o \rfloor$.

Proposition 2 states the state variation correctness, by claiming that changing an input value and executing a block with a state variation leads us to a state variation.

Proposition 2 (State variation correctness). *Let* B *be an instruction block. For all concrete states* Q_1, Q_1', *for all abstract state* $\overline{Q_1}$, *for all* Γ_1,

If $Q_1' \overset{\overline{Q_1},\beta}{\longleftrightarrow} Q_1$ *then* $\mathit{instr}_B(Q_1') \overset{\overline{\mathit{instr}_B}(\overline{Q_1},\Gamma_1),\beta}{\longleftrightarrow} \mathit{instr}_B(Q_1)$.

We can now conclude with the proof of the non-interference theorem.

Proof of Theorem 1. Let B be an instruction block. Let $Q_1 = ((i, \rho_1, s_1) :: fr_1, G_1)$ be a concrete state, $\overline{Q_1} = (\overline{\rho_1}, \overline{s_1}, \overline{G_1})$ such that $Q_1 \lhd^{\alpha} \overline{Q_1}$, $Q_2 = \mathit{instr}_B(Q_1)$, and $\overline{Q}_2 = \overline{\mathit{instr}_B}(\overline{Q}_1, \Gamma_1)$. Let $o \in \mathcal{O}(G_1)$ be an object node and $\iota \in \mathcal{I}(Q_1)$ be an input value. Let us consider a value-change $Q_1' = ((i, \rho_1', s_1) :: fr_1, G_1')$ of Q_1 and ι. According to Definition 2, either ι is a value node of G_1 and then $G_1' = G_1[\iota \mapsto v]$, or ι is a local variable and then $\rho_1' = \rho_1[\iota \mapsto v]$ (for some $v \in \mathit{Val}$). Let us denote $\beta = \alpha_{Q_1}^{\overline{Q_1}}(\iota)$, then, the first case corresponds to the graph variation rule in Definition 8, the second case to the local variables variation rule of Definition 8, thus $Q_1' \overset{\overline{Q_1},\beta}{\longleftrightarrow} Q_1$. We denote $Q_2' = \mathit{instr}_B(Q_1')$. According to Proposition 2, we have $Q_2' \overset{\overline{Q_2},\beta}{\longleftrightarrow} Q_2$. As $\mathit{free}_{\alpha}(\alpha(o), \alpha_{Q_1}^{\overline{Q_1}}(\iota), Q_1)$, graph variation rule of Definition 8 says that $o \in \mathcal{O}(G_1') \cap \mathcal{O}(G_2')$. As by hypothesis, $\mathit{Reach}_{\overline{G_2}}(\alpha(o) \cap \alpha_{Q_1}(\iota)) = \varnothing$, by applying Lemma 1, we obtain $G_2\lfloor o \rfloor \equiv G_2'\lfloor o \rfloor$ (for G_2, G_2' the memory graphs of resp. Q_2, Q_2'), ie ι does not interfer with o in B. □

6 Inter-method Analysis

In this section, we add a support for a context-insensitive or a context-sensitive analysis with the same semantics rules depending on the inter-procedural control flow graph.

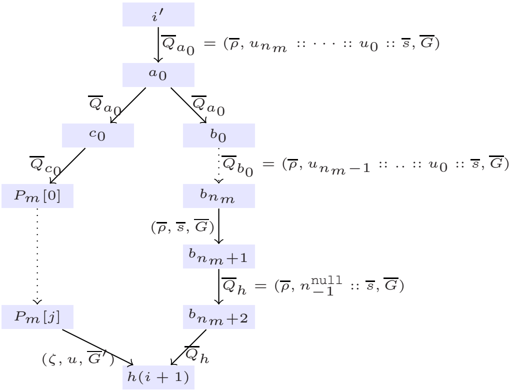
In programs with method call, the size of the call stack in the concrete execution is not bounded: this is one of the main problems for abstract interpretation as it deals only with finite abstract domains. Some works of the literature propose an abstraction of the

```
$i$       : invoke m
$i+1$ : ...
```

(a) Original code

```
$i$                 : iconst_0
$i'$                : iconst_0
$a_0$               : ifcmp $c_0$
$b_0$               : pop
                    : ...
$b_{n_m}$           : pop
$b_{n_m+1}$         : aconst_null
$b_{n_m+2}$         : goto $h(i+1)$
$c_0$               : invoke m
$h(i+1)$            : ...
```

(b) Transformed code


i'
$\overline{Q}_{a_0} = (\overline{\rho}, u_{n_m} :: \cdots :: u_0 :: \overline{s}, \overline{G})$
a_0
$\overline{Q}_{a_0}$
$\overline{Q}_{a_0}$
c_0
b_0
$\overline{Q}_{c_0}$
$\overline{Q}_{b_0} = (\overline{\rho}, u_{n_m-1} :: .. :: u_0 :: \overline{s}, \overline{G})$
$P_m[0]$
b_{n_m}
$(\overline{\rho}, \overline{s}, \overline{G})$
b_{n_m+1}
$\overline{Q}_h = (\overline{\rho}, n^{\texttt{null}}_{-1} :: \overline{s}, \overline{G})$
$P_m[j]$
b_{n_m+2}
$(\zeta, u, \overline{G'})$
$\overline{Q}_h$
$h(i+1)$

where $\overline{Q}_{c_0} = (\{0 \mapsto u_0, \ldots, n_m \mapsto u_{n_m}\}, \epsilon, \overline{G})$

(c) Transformed Control Flow

Fig. 6. Bytecode transformation

call stack [14]. We follow a different direction and propose an analysis *without any abstraction of the call stack*: we are going to keep the same abstract domain than in the intra-method analysis, in which we occult the call stack and only consider the current frame of the method: the method invocation can be added to the framework used. To do so, we use a code transformation: we replace each `invoke` bytecode by an "`if`" region as depicted in Figures 6(a) and 6(b) and we add the following convenient abstract semantics rules for `invoke` and `areturn` bytecodes (where ζ is a special value) :

$$\frac{(\rho, u_{n_m} :: \cdots :: u_0 :: s, G)}{(\{0 \mapsto u_0, \ldots, n_m \mapsto u_{n_m}\}, \epsilon, G)}\,\texttt{invoke m} \qquad \frac{(\rho, u :: s, G)}{(\zeta, u, G)}\,\texttt{areturn}$$

The invocation of a method implies the creation of a new frame, the old one called the calling context being preserved. Conversely, the return of the method yields the destruction of the current frame, the calling context being resumed. Obviously, the two abstract rules given above do not mimick this process. This is the role devoted to the program transformation by means of its abstraction.

This transformation is syntactical and harmless: the path $c_0 P_m[0] \ldots P_m[j]$ corresponds to method invocation, while the path $b_0 \ldots b_{n_m+2}$ is dead code [3] and is used to "transmit" the context under which the method was invoked to the immediate successor of the invocation, thus "hiding" the context save and reload that occur the concrete execution of an invocation. The equivalent of a context reload is done on abstract states by the join operator when computing $\overline{Q}_{h(i+1)}$. We extend the operator to the case where the local variables have the special value ζ: $(\zeta, u, G) \sqcup (\rho, v :: s, G') = (\rho, v :: s, G') \sqcup (\zeta, u, G) = (\rho, u :: s, G \sqcup G')$, thus as $G \sqcup G' = G'$, $\overline{Q}_{h(i+1)} = (\rho, u :: s, G')$

[3] However, note that this code perserves usual Java properties: the stack is well typed and has always a fixed size at each program point.

which contains the local variables of $\overline{Q}_{a_0}$, the stack of $\overline{Q}_{a_0}$ increased with the return value of m, and the memory resulting from the execution of m.

Soundness. In order to show that the extended abstract model is sound with respect to the non-interference, we prove that the abstract semantics rules for invoke and areturn have the same properties as the rest of bytecode. We only have to define an extension of the relation $\lhd^{\alpha}$ for an abstraction relation α taking into account the new value ζ. This extension is necessary for the areturn bytecode, in order to prove the relation $\lhd^{\alpha}$ for the value returned by the invoked method. This definition is sufficient, since the relation $\lhd^{\alpha}$ for the rest of the calling context is being insured by the code transformation. For the same reasons, we need to define an extension for the state variation definition: for the areturn bytecode, and thus the special value ζ, the state variation must hold only for the returned value.

7 Secure Information Flow

Most of the previous works on non-interference require the lattice security model of information flow to be known from the beginning. The AMG contains points-to and control flow dependency information. Once computed, the AMG can be labeled with security levels and non-interference checked. We can consider any kind of "programs", let us for example consider a method m (with n parameters), for which we want to check non-interference. We have to define a initial abstract state $\overline{Q}^{\mathtt{m}}_{init} = (\{0 \mapsto \{n^p_0, \mathtt{d}\}, \ldots, n \mapsto \{n^p_n, \mathtt{d}\}\}, \epsilon, \overline{G}_{init})$ for the analysis such that $\overline{G}_{init} = (\overline{V}_{init}, \overline{E}_{init})$ and $\overline{V}_{init}$ contains all the nodes required by the allocation site model, constants and null value, and nodes representing formal parameters. For each parameter i, n^p_i denotes the i^{th} parameter of the method. We add to $\overline{G}_{init}$ the minimal subgraph rooted by n^p_i, $\overline{G}^p_i = (\overline{V}^p_i, \overline{E}^p_i)$. This graph represents the parameter and all of its content. To avoid infinite graphs, we use a parameter h, which is called the height of the analysis, and represents how deep we unfold the recursive data structures. The graph $\overline{G}^p_i$ is the minimal graph that contains n^p_i such that $\forall u \in \mathcal{O}(\overline{V}^p_i)$, for each field f_1 of $Type(u)$:

- either there exists a path $u^1_1 u^1_2 \ldots u^1_k u^2_1 u^2_2 \ldots u^2_k \ldots u^h_1 u^h_2 \ldots u^h_k u$ labeled by $(\langle f_1, \mathtt{d}\rangle\langle f_2, \mathtt{d}\rangle\langle f_3, \mathtt{d}\rangle \ldots \langle f_k, \mathtt{d}\rangle)^h$ in $\overline{G}^p_i$ such that $\forall 1 \leq i \leq h, 1 \leq j \leq h$, $Type(u^i_l) = Type(u^j_l)$, then $(u, u^h_1, \langle f_1, \mathtt{d}\rangle) \in \overline{E}^p_i$,
- or $u.f_1 \in \overline{V}^p_i$ and $(u, u.f_1, \langle f, \mathtt{d}\rangle) \in \overline{E}^p_i$.

Then, we use that description of "formal" initial state. Let $(\mathcal{L}, \sqcup, \sqsubseteq)$ be a lattice of security levels; for example, $\mathcal{L} = \{low, high\}$. We can now define security levels on types: let $\lambda_t : Fields \times Classes \rightarrow \mathcal{L}$ be a function that associates a security level with a field of a class. A security function is a function that associated security levels to input values and to return values (denoted by the set Ret): $\lambda_i : \bigcup_{0 \leq i \leq n} \mathcal{V}(\overline{V}^p_i) \cup Ret \rightarrow \mathcal{L}$ such that if v is the field f of a class C, then $\lambda_i(v) = \lambda_t(f, C)$.

We say that a method is secure if values accessible from parameters or return value contain at least a field of higher security level than their own on the path.

Theorem 2 (Secure information flow). *Let* m *be a method, and* λ_i *a security function. Let* $\overline{Q} = (\overline{\rho}, \overline{u} :: \overline{s}, (\overline{V}, \overline{E})) = \overline{instr}_m(\overline{Q}_{init}, \epsilon)$*, and* $Ret = \{v \in \overline{V} \mid \exists(v,t) \in \overline{u}\}$*. The method has secure information flow with respect to* $\mathcal{L}$ *if for every node* $v \in \mathcal{V}(\overline{V})$ *and every path* $o_0 \overset{\langle f_1, t_1 \rangle}{\rightarrow} o_1 \ldots o_k \overset{\langle f_k, t_k \rangle}{\rightarrow} o_{k+1}$ *with* $o_0 \in Ret \cup \{v \in \overline{V} \mid v = n^s_{-1} \lor v = n^p_i\}$*,* $\exists i$ *such that* $\lambda_i(v) \sqsubseteq \lambda_t(f_i, Type(o_i))$*.*

For example, we can define two security policies for the graph in Figure 4: *(i)* $\lambda_t(sal, Income) = high$ and $\lambda_t(tax, Income) = high$ and *(ii)* $\lambda_t(sal, Income) = high$ and $\lambda_t(tax, Income) = low$. In the first case, the program is secure, while in the second case the program is insecure as there is an edge $(p_0, p_0.sal, \langle tax, \mathtt{i} \rangle)$ and $\lambda_i(p_0.sal) \sqsupseteq \lambda_t(tax, Income)$.

The advantage of our approach is that security annotations must not be known a priori. Changing a security level does not require a new analysis. Note that we can also have more precise policies on instances: if o and o' have the same type, $o.f$ and $o'.f$ can be given different security levels using a function $\lambda_e : \overline{E}_{init} \rightarrow \mathcal{L}$ instead of λ_t. It is more precise but require the user to precise all the policies.

8 Conclusion

Information flow analysis aims to avoid that programs leak confidential information: observable/public outputs of a program must not disclose information about secret/private values manipulated by the program. Motivated by information flow analysis for Java bytecode, we propose an algorithm to compute AMGs, which tracks, at different program points, how input values may influence the outputs. Such a graph is a points-to graph extended with primitive values and flows arising from the control structure of a program. We prove the soundness of our construction by a non-interference theorem: if a node a is related to a node b then the value of b may influence the value of a.

In contrast with usual information flow techniques, our approach is flow-sensitive and the security levels are not required to be known during the graph computation. Changing a security level does not require reanalyzing the code. Our main goal is to provide an algorithm which can be use for information flow. In the same time, as the AMG includes the points-to graph, our framework can be exploited for any points-to application in program analysis, such as escape analysis and optimization.

References

1. Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A core calculus of dependency. In: Proc. POPL 1999, pp. 147–160. ACM Press, New York (1999)
2. Amtoft, T., Bandhakavi, S., Banerjee, A.: A logic for information flow in object-oriented programs. ACM SIGPLAN Notices 41(1), 91–102 (2006)
3. Avvenuti, M., Bernardeschi, C., De Francesco, N.: Java bytecode verification for secure information flow. ACM SIGPLAN Notices 38(12), 20–27 (2003)
4. Ball, T.: What's in a region?: or computing control dependence regions in near-linear time for reducible control flow. ACM Letters on Programming Languages and Systems 2(1-4), 1–16 (1993)

5. Banerjee, A., Naumann, D.A.: Secure information flow and pointer confinement in a java-like language. In: Proc. IEEE CSFW 2002, Washington, DC, USA, p. 253 (2002)
6. Barthe, G., Pichardie, D., Rezk, T.: A certified lightweight non-interference java bytecode verifier. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 125–140. Springer, Heidelberg (2007)
7. Denning, D.E.: A lattice model of secure information flow. Communications of the ACM 19(5), 236–243 (1976)
8. Genaim, S., Spoto, F.: Information Flow Analysis for Java Bytecode. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 346–362. Springer, Heidelberg (2005)
9. Ghindici, D., Grimaud, G., Simplot-Ryl, I.: Embedding verifiable information flow analysis. In: Proc. PST 2006, Toronto, Canada, November 2006, pp. 343–352 (2006)
10. Ghindici, D., Simplot-Ryl, I., Talbot, J.-M.: A sound dependency analysis for secure information flow (extended version). Technical Report 0347, INRIA (November 2007)
11. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy, pp. 11–20 (1982)
12. Hammer, C., Krinke, J., Snelting, G.: Information flow control for java based on path conditions in dependence graphs. In: Proc. IEEE ISSSE 2006, pp. 87–96 (2006)
13. Hunt, S., Sands, D.: On flow-sensitive security types. In: Proc. POPL 2006, pp. 79–90. ACM Press, New York (2006)
14. Jeannet, B., Serwe, W.: Abstracting call-stacks for interprocedural verification of imperative programs. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 258–273. Springer, Heidelberg (2004)
15. Kobayashi, N., Shirane, K.: Type-based information flow analysis for low-level languages. In: Proc. APLAS 20O2, Shanghai, China, pp. 302–316 (2002)
16. Myers, A.C.: Jflow: practical mostly-static information flow control. In: Proc. POPL 1999, pp. 228–241. ACM Press, New York (1999)
17. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)
18. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
19. Salcianu, A., Rinard, M.: Pointer and escape analysis for multithreaded programs. ACM SIGPLAN Notices 36(7), 12–23 (2001)
20. Sun, Q., Banerjee, A., Naumann, D.A.: Modular and constraint-based information flow inference for an object-oriented language. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 84–99. Springer, Heidelberg (2004)
21. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security 4(2-3), 167–187 (1996)
22. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: Proc. OOPSLA 1999, Denver. ACM SIGPLAN Notices, vol. 34, pp. 187–206 (1999)

Refinement Patterns for Hierarchical UML State Machines

Jens Schönborn[1,⋆] and Marcel Kyas[2,⋆⋆]

[1] Christian-Albrechts-Universität zu Kiel, Germany
jes@informatik.uni-kiel.de
[2] Department of Computer Science, Freie Universität Berlin, Germany
marcel.kyas@fu-berlin.de

Abstract. While the semantics of (labeled) transition systems and the relations between these are well understood, the same still needs to be achieved for UML 2.x state machines, because the UML standard is ambiguous and admits many semantics, which are often defined in terms of labeled transition systems.

A formal semantics for UML state machines with interlevel transitions and notions of refinement are described to enable the study of *transformations*, i.e., functions from state machines to state machines, and to establish the conditions under which these transformations are refinement steps. Many of these transformations are described and shown to be refinement steps. A language extension is finally proposed that help modelers to ensure that all transformations are indeed refinements.

1 Introduction

Automaton-based languages are popular for modeling systems, because they are intuitively understood. The semantics of and the relations between flat transition systems are well understood. Languages of hierarchic automata, like StateCharts [1] and their object-oriented variant [2], ROOM [3], Argos [4], and UML 2.x state machines [5], provide structuring mechanisms and concise notations. The increased expressive power gained by this is at the expense of accessibility: understanding the meaning of a state machine is often hard and requires expert knowledge. The semantics of and especially the relations between these hierarchical UML state machines, which we analyze in this paper, are not yet understood completely. This is caused by the underspecified standard [6] and by the complexities introduced by hierarchy and *interlevel* transitions, i.e., transitions between different levels of the hierarchy. Many semantics for UML state machines have been proposed in the literature, e.g., [7, 8, 9, 10, 11, 12], which often differ in the way fireable transitions are chosen. Our language of UML state

[⋆] J. Schönborn's work has been supported by DFG-project FE 942/1-1 RO 1122/12-2 *refism* (http://www.informatik.uni-kiel.de/~refism/refism.html).

[⋆⋆] M. Kyas' work has been supported by EU-project IST-33826 *CREDO: Modeling and analysis of evolutionary structures for distributed services* (http://credo.cwi.nl).

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 371–386, 2010.

machines and its semantics follows [12], which we believe to formalize state hierarchy and interlevel transition [5] most faithfully, and is summarized in Sect. 2.

The above mentioned loss of accessibility becomes apparent in top-down development methods, where the desire is to *refine* a first model of a system by a series of design steps, which preserve all desired properties and eliminate undesired behaviors, with the intention to arrive at a final model that is executable. The compact notation of UML state machines makes it hard to relate the syntactic descriptions of the model: hierarchy and priorities have undesired effects on refinement. Often changes can be made liberally, since the priority rules will disregard these changes. Other changes may have far reaching consequences, since the newly defined behavior can eliminate existing behavior completely due to priorities. Finally, simple steps like removing a transition may just *change* the behavior instead of restricting it. The notion of refinement used in this paper is described in Section 3 and follows David Park [13].

This paper concerns "patterns of refinement." These are syntactic transformations of a state machine that are in effect refinements. Syntactic patterns can be supported by tools, which can check syntactic side conditions, and allow modelers to refine the system easily, because they are not concerned with proving validity of the refinement steps. Many patterns have been proposed in literature for flat state machines [14] or for UML state machines with different priority rules [15]. Our refinement patterns are described in Section 4. Previously described patterns are shown to be invalid in our semantics and others have been adapted. The formalization of *redefinable*, a predicate on states that enables transformations on state machines, is the main technical contribution. It makes the context of underspecification explicit.

We go first steps towards a theory of refinement, which may be integrated into tool-supported, UML based development processes, with our semantics, a formulation of refinement for UML state machines, and refinement patterns.

To conclude, we compare our refinement patterns to previously published patterns in Section 5. The nature of UML's choice of priorities introduces complex side conditions, which are avoided by other researchers, because they either do not consider hierarchy at all or use non-standardized priorities that simplify their calculus, effectively creating a language not defined by the OMG.

2 UML State Machines

UML State Machines are a variant of Harel's Statecharts [1]. The most important difference is, that UML state machines use a *run-to-completion* step semantics: One single event is selected and the reaction is computed up to the point the system stabilizes. Remaining events are considered for future transitions. Harel's Statecharts use all events for the transition, and the reaction will try to use the largest set of transitions.

Many assumptions and transformations of the Statechart-like languages do not hold for UML state machines. For example, interlevel transitions cannot be eliminated from UML state machines like in Argos [4], because this changes their semantics.

We focus on refinements patterns for *hierarchical* state machines with *inter-level* transitions here. We use a subset of the state machines syntax and semantics defined in [12], which we summarize below. Object and variable states are not considered because they may be encoded in the state machines states and events. Actions remain uninterpreted and are observations of the system.

2.1 Syntax

The basic concepts of UML 2.0 state machines are transitions and states between transitions. States form a hierarchy: A state may contain regions, which are separated by dashed lines and each region may again contain states such that the resulting structure forms a tree with a region, here always called ϵ as root. For the sake of brevity we omit most pseudo states, especially fork- and join-pseudostates, as well as entry-, exit-, and do-activities. The semantics of completion events is a subtle and imprecise matter [6]. We therefore omit them.

Finally, we do not treat variables here. The states that arise from variables and their assignments could be encoded into the states of the state machine itself, whereas the values received by events can be encoded by creating enough events to encode these values. Moreover, the theory presented in this paper is not concerned with the state that arises from variables and their assignments but with the graphical notation of state machines. Variables and their assignments are best treated by traditional data refinement techniques [16], which may well complement our method. Consequently, we only focus on events that trigger transitions and use actions as uninterpreted observations of the state machines reaction, as we show later. Our mathematical notation follows Davey and Priestley [17] and all concepts are summarized in Table 1.

Definition 1 (State Hierarchy). *Let V be a set of* vertexes, *partitioned into* states S *and* regions R *with the* root region $\epsilon \in R$. *The function* $\mathsf{prt} : R \setminus \{\epsilon\} \to S + S \to R$, *read* parent, *maps states to their containing regions, and regions to their containing states such that the derived containment relation $\succ$, defined as the transitive closure of $\{(z, \mathsf{prt}(z)) \mid z \in S \cup R \setminus \{\epsilon\}\}$, is irreflexive, i.e. a tree with root ϵ. The relation $\succeq$ denotes the reflexive closure of $\succ$. Then (S, R, prt) is called a* state hierarchy. *A state s is called* basic *state if $\forall r \in R : \mathsf{prt}(r) \neq s$. A state hierarchy is called* flat, *if for all $\forall s \in S : \mathsf{prt}(s) = \epsilon$. The* downset $\downarrow(v) \triangleq \{v' \in V \mid v \succeq v'\}$ *denotes all vertexes below v. Similarly, the* upset $\uparrow(v) \triangleq \{v' \in V \mid v' \succeq v\}$ *denotes all vertexes above v. Furthermore the downset resp. upset without the vertex itself is $\downharpoonleft(v) \triangleq \downarrow(v) \setminus \{v\}$ resp. $\upharpoonright(v) \triangleq \uparrow(v) \setminus \{v\}$.*

Configurations describe snapshots of a state machine's execution. They consist of the active states. We call a region, different from the root region, active if and only the containing state is active. At any time, the root region is active. For any active non empty region there is exactly one active state in that region and for all active states all regions in these states are active. Formally, a configuration is a subtree of the state hierarchy, as defined below. Each state machine will start in an initial configuration.

Definition 2 (Configuration). *The set of all configurations $\mathcal{C}$ of a state hierarchy (S, R, prt) is a set of sets of states $\mathcal{S} \subseteq S$, where the sets of states satisfy the following well-formedness conditions:*

1. $\exists s \in \mathcal{S} : \mathsf{prt}(s) = \epsilon$
2. $\forall r \in R \setminus \{\epsilon\} : \mathsf{prt}(r) \in \mathcal{S} \wedge (\exists s \in S : \mathsf{prt}(s) = r) \implies (\exists s \in \mathcal{S} : \mathsf{prt}(s) = r)$
3. $\forall s, s' \in \mathcal{S} : s \neq s' \implies \mathsf{prt}(s) \neq \mathsf{prt}(s')$

Transitions are triggered by an *event*, e.g., a message the machine is receiving. If a transition is taken, then an action is performed as part of the reaction.

Definition 3 (Transitions). *A* transition $t \triangleq (s, e, \psi, s')$ *is a tuple, in which $s, s' \in S$ s called* source state *and s' called* target state. *Event $e \in E$ is called its* trigger *(which has to be provided to enable the transition), and E denotes the set of all events. $\psi \in \Psi$ is called its action, where Ψ is a set of uninterpreted actions that contains the special 'do nothing'-action* skip. *The set of all transitions is denoted by T and projections of transitions to the corresponding components are denoted by $\pi_{\text{sor}}(t)$, $\pi_{\text{ev}}(t)$, $\pi_{\text{act}}(t)$, and $\pi_{\text{tar}}(t)$, respectively.*

Suppose that there is state s from which no transition emanates which is triggered by some event e. According to the UML semantics, e is consumed without any other effect (except when e is deferrable). This default behavior interferes with any refinement method, because the modeler cannot add a state and it's outgoing transitions in two refinement steps.

Instead, we allow the modeler to change UML's default behavior by declaring an event *redefinable* for a state: e is consumed but the reaction is chaotic: any well-formed configuration and any multiset of actions may be generated as a reaction. Then adding transitions emanating s preserves behavior and is a refinement step. Observe that unlike in Rumpe's work [14], adding a transition does not restrict behavior, thus prohibiting the introduction of further transitions triggered by the same event, for the outgoing event, such that more transitions for the same event may be added in different steps.

Definition 4 (Redefinable). *The predicate* $\mathsf{redef}(e, s)$ *indicates that state s is still redefinable with respect to the event e. As long as a state is* redefinable*, all behavior triggered by e can emanate from s, which models the effect of possible future redefinitions. A redefinable state is written «redef E», where E enumerates all events that are redefinable.*

Hierarchical states require us to consider the property of redefinable of the containing states when adding a new state as exemplified in the following.

Example 1. Suppose a state machine as depicted in Fig. 1 on the left where no states are marked redefinable. Now we add a new state s_0 as substate of s_1 which redefinable wrt. event e and a transition from s_0 to s_3 as depicted in Fig. 1 on

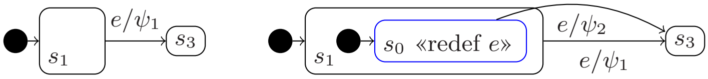


Fig. 1. State machine before (left) and after (right) refinement

Table 1. Notation

Model elements	
$v \in V$	vertexes
$s \in S$	states
$r \in R$	regions
$\epsilon \in R$	root region
$e \in E$	events
$\psi \in \Psi$	actions
skip	'do nothing'-action
$t \in T$	transitions
$\mathcal{S} \in \mathcal{C}$	configurations
$\mathcal{S}^i \in \mathcal{C}$	initial configuration

Functions	
$s \succ s'$	s' is a proper substate of s
$s \succeq s'$	$s \succ s'$ or $s = s'$
$\mathsf{prt}(v)$	Parent vertex of v
$\uparrow(v)$	upset $\{v' \mid v' \in V \land v \succeq v'\}$.
$\downarrow(v)$	downset $\{v' \mid v' \in V \land v' \succeq v\}$.
$\mathsf{default}(r)$	Default state in region r
$\mathsf{redef}(e, s)$	State s is redefinable for event e
$\pi_{\text{sor}}(t)$	Source state of transition t
$\pi_{\text{ev}}(t)$	Trigger of transition t
$\pi_{\text{act}}(t)$	Action of transition t
$\pi_{\text{tar}}(t)$	Target state of transition t

the right. We would now observe behavior ψ_2 rather than ψ_1 as we have expected it from the facts that s_1 already has an outgoing transition triggered by e and it is not marked redefinable wrt. to e. Therefore we require:

$$\forall e \in E : \forall s \in S : \mathsf{redef}(e, s) \implies \forall s' \in \downarrow(s) \cap S : \mathsf{redef}(e, s') \tag{1}$$

Definition 5 (State Machines). *A* state machine *SM is a tuple* $((S, R, \mathsf{prt}), T, \mathsf{default}, \mathsf{redef}, \mathcal{S}^i)$*, where* (S, R, prt) *is a state hierarchy,* $\mathsf{default} : R \to S$ *assigns to each non empty region a* default *state, T is a set of transitions, and $\mathcal{S}^i \in \mathcal{C}$ its initial configuration.*

2.2 Operational Semantics

In this section we define the semantics of a UML state machines. We loosely follow [12], where a detailed semantics of UML state machines is described. The execution proceeds as follows: After selecting an event from an input pool, the state machine selects a maximal conflict-free set of transitions, moves to the new configuration, and outputs all actions as part of the effect. A transition may leave and enter *many* states at once, because the states are structured hierarchically. We have to compute the set of states a transition is leaving and the set of states it is entering. This is computed using the *least common region*:

Definition 6 (LCR). *Let (S, R, prt) be well-formed. The* least common region *is the lowest region which contains all of a given subset of the states $S' \subseteq S$.*

$$\mathsf{lcr}(S') \triangleq r \iff r \in R \land (\bigwedge_{s \in S'} s \succ r) \land (\forall r' \in R : \bigwedge_{s \in S'} s \succ r' \implies r \succeq r')$$

Since any state hierarchy is a tree with root ϵ, $\langle V, \succeq \rangle$ induces a join-semilattice with top element ϵ. We observe the following properties: (1) $v \sqcup v' \in R$ if $v \neq v'$ and $v, v' \in S$, and (2) $v \sqcup v' \in S$ if $v \neq v'$ and $v, v' \in R$. Consequently, lcr is a well-defined function.

Two transitions are in conflict, if they leave a common state. A transition t leaves all states of a configuration $\mathcal{S}$ that are below the least common region of both its source and target state. This is formalized by $\uparrow(\mathsf{lcr}(\{\pi_{\text{sor}}(t), \pi_{\text{tar}}(t)\}))\cap\mathcal{S}$.

Definition 7 (Conflicting Transitions). *The set of pairs of transitions that* conflict *in configuration $\mathcal{S}$ is:*

$$\mathsf{conflict}(\mathcal{S}) \triangleq \{(t, t') \in T \times T \mid \mathcal{S} \cap (\uparrow(\mathsf{lcr}(\{\pi_{\text{sor}}(t), \pi_{\text{tar}}(t)\}))) \cap (\uparrow(\mathsf{lcr}(\{\pi_{\text{sor}}(t'), \pi_{\text{tar}}(t')\}))) \neq \emptyset\} \quad .$$

Example 2. The two transitions labeled e/ψ_1 and e/ψ_2 of right state machine in Fig. 1 conflict when event e is dispatched end states s_0 and s_1 are active.

Definition 8 (Enabled Transitions). *A transition is* enabled *for trigger $e \in E$ and active states $\mathcal{S}$ if the source state is active and if the trigger of the transition is e.* $\mathsf{enabled}(e, \mathcal{S}) \triangleq \{t \in T \mid \pi_{\text{sor}}(t) \in \mathcal{S} \wedge \pi_{\text{ev}}(t) = e\}$

When transitions are in conflict, they are selected by priority. A transition whose source is more deeply nested has higher priority. By defining priority on states rather than on transitions we are able to consider redefinable states. A state s has priority over state s' if and only if: $s \succ s'$.

While a state machine will try to execute as many *fireable* transitions as possible for a given event, there might be many solutions to those sets.

Definition 9 (Fireable Transitions). *A set of transitions is* fireable *if it is a non empty maximal set of enabled and conflict-free transitions such that no enabled transition which is not in the set and with higher priority exists.*

$$\begin{aligned} \mathsf{fireable}(e, \mathcal{S}) \triangleq \big\{T' \subseteq \mathsf{enabled}(e, \mathcal{S}) \mid T' \neq \emptyset \wedge \\ \big(\forall t, t' \in T' : (t, t') \in \mathsf{conflict}(\mathcal{S}) \implies t = t'\big) \wedge \big(\forall t \in \mathsf{enabled}(e, \mathcal{S}) \setminus T' : \\ \big(\forall t' \in T' : \neg(\pi_{\text{sor}}(t) \succ \pi_{\text{sor}}(t')) \wedge \exists t' \in T' : (t, t') \in \mathsf{conflict}(\mathcal{S})\big)\big)\big\} \end{aligned}$$

Definition 10 (Update Configuration). *Define the update of a state configuration $\mathcal{S}$ with respect to a transition t as*

$$\mathsf{upd}(t, \mathcal{S}) \triangleq \mathcal{S} \setminus \uparrow(\mathsf{lcr}(\{\pi_{\text{sor}}(t), \pi_{\text{tar}}(t)\})) \cup \mathsf{enter}(\mathsf{lcr}(\{\pi_{\text{sor}}(t), \pi_{\text{tar}}(t)\}), \pi_{\text{tar}}(t)) \quad ,$$

where

$$\mathsf{enter}(v, s) \triangleq \begin{cases} \{v\} \cup \bigcup_{r \in R \wedge \mathsf{prt}(r) = v} \mathsf{enter}(r, s) & \textit{if } v \in S \\ \emptyset & \textit{if } v \in R \setminus \textit{dom}(\mathsf{default}) \\ \mathsf{enter}(\mathsf{default}(v), s) & \textit{if } v \in (R \cap \textit{dom}(\mathsf{default})) \setminus \downarrow(s) \\ \mathsf{enter}(\mathsf{next}(v, s), s) & \textit{otherwise,} \end{cases}$$

where $\mathsf{next}(v, s) \triangleq s' \in\downarrow(s) \cap \uparrow(v) \iff \forall s'' \in\downarrow(s) \cap \uparrow(v) : s'' \succeq s'$. *Define the update of a state configuration with respect to a set of transitions*

$$\mathsf{upd}(T', \mathcal{S}) \triangleq \begin{cases} \mathcal{S} & \text{if } T' = \emptyset \\ \mathsf{upd}(T' \setminus \{t\}, \mathsf{upd}(t, \mathcal{S})) & \text{for some } t \in T'. \end{cases}$$

Proposition 1. *The configuration* $\mathsf{upd}(T', \mathcal{S})$ *is uniquely determined for any set* $T' \in \mathsf{fireable}(e, \mathcal{S})$.

Proof. Computing $\mathsf{upd}(T', \mathcal{S})$ is confluent, i.e., the order in which transitions are chosen from T' is irrelevant, because $T' \in \mathsf{fireable}(e, \mathcal{S})$ is conflict free. □

Definition 11 (Effect). *Let* T' *be a set of transitions. Its* effect $\pi_{\text{act}}(T')$ *is the* multiset $\wr \pi_{\text{act}}(t) \mid t \in T' \wr$. *(We write* $M^{\mathbb{N}}$ *for the set of all multisets over* M*.)*

The effect of a transition is a multiset, because we do not consider the states of the variables, where the order of executing the actions might have an observable effect, and because the different orders of executing actions is an unnecessary complication. Instead, we could have chosen sequences of actions, with transitions that only differ in the order of actions.

As long as a state is marked redefinable wrt. some event arbitrary behavior may occur upon receiving that event. This anticipates all future changes to the state that involves transitions triggered that event.

Definition 12 (Chaotic Behavior). *We have* chaotic behavior *if there is a currently active state in configuration* $\mathcal{S}$ *that is still marked redefinable with respect to event e and there exists a set of fireable transitions such that all transitions in that set do not have higher priority.*

$$\begin{aligned} univ(e, \mathcal{S}) \iff & \exists s \in \mathcal{S} : \mathsf{redef}(e, s) \wedge \\ & \big(\mathsf{fireable}(e, \mathcal{S}) = \emptyset \vee \exists T' \in \mathsf{fireable}(e, \mathcal{S}) : \forall t \in T' : \neg(\pi_{\text{sor}}(t) \succ s)\big) \end{aligned}$$

Definition 13 (Structural Operational Semantics). *The operational semantics of a state machine is given by the rules below:*

$$\frac{T' \in \mathsf{fireable}(e, \mathcal{S})}{\mathcal{S} \xrightarrow{e/\pi_{\text{act}}(T')} \mathsf{upd}(T', \mathcal{S})}\ \text{STEP} \qquad \frac{univ(e, \mathcal{S}) \quad v \in \Psi^{\mathbb{N}} \quad \mathcal{S}' \in \mathcal{C}}{\mathcal{S} \xrightarrow{e/v} \mathcal{S}'}\ \text{CHAOS}$$

$$\frac{\mathsf{fireable}(e, \mathcal{S}) = \emptyset \quad \{s \in \mathcal{S} \mid \mathsf{redef}(e, s)\} = \emptyset}{\mathcal{S} \xrightarrow{e/\mathsf{skip}} \mathcal{S}}\ \text{DISCARD}$$

$\rightarrow^*$ *denotes the transitive closure of the defined transition relation.*

The structural operational semantics gives rise to a new transition system. For a state machine SM we refer to its semantic counter-part as TS. The set of all traces of a state machine, written $[\![SM]\!]$, is a sequence of possibly infinite sequences of pairs e/ψ.

Note that actions can be included into the semantics as follows: Change the update function upd such that the effect of a single transition is applied to both a configuration and a variable assignment. Now use configurations together with variable assignments as the SOS transitions system states and adapt the SOS-rules accordingly.

3 Refinement and Simulation

The UML 2.x standards use the term *refinement* or its opponent *abstraction* in various locations without a specified meaning. The closest characterization is [5, p. 696]:

> Specifies a refinement relationship between model elements at different semantic levels, such as analysis and design. The mapping specifies the relationship between the two elements or sets of elements. The mapping may or may not be computable, and it may be unidirectional or bidirectional. Refinement can be used to model transformations from analysis to design and other such changes.

This characterization does not specify what is actually meant by refinement. It serves to document development steps but does not specify what the relation between two models is. This relation is a "semantic variation point."

On the other hand, *refinement* has a very specific sense in literature (see, e.g., [16] for references) and relates the behavior of two entities. The intuition is that the more specific entity C can substitute for the more abstract one A in a context $\mathtt{C}$ in such a way that $\mathtt{C}$ cannot distinguish C from A.

A refinement in the sense of [5, p. 106] is not a behavioral refinement, because UML 2.x allows for the refining operation to "add new preconditions and postconditions" whereas behavioral refinement would only allow to *remove* preconditions.

Refinement relations are often understood as *pre-orders* $\sqsubseteq$ on systems, where $M \sqsubseteq M'$ is read as: All observations made about M are also observations made about M'. Observational equivalence $M \equiv M'$ can then be understood as $M \sqsubseteq M' \wedge M \sqsubseteq M'$.

A transformation is a mapping T from state machines M to state machines $T(M)$. Such transformations can be formalized as graph transformations, that just change a part of the state machine and leave the remainder invariant.

Subsequently, we study transformations on UML 2.x state machines and characterize the semantics preserved by them. A transformation T preserves behavior, if and only if $T(M) \sqsubseteq M$ holds for all state machines M. It is well known, that different observations allow for different semantics and that the semantics form a *lattice*: Graph isomorphism is the strongest congruence, mutual trace inclusion the weakest one [18].

Bernhard Rumpe [14] defines *behavioral refinement* by trace-inclusion, i.e., an automaton C refines an automaton A, if all runs of C are also runs of A. For the purpose of this paper we will use the stronger notion of *simulation* [13]. We have to adapt the usual definition of simulation to the predicate redefinable.

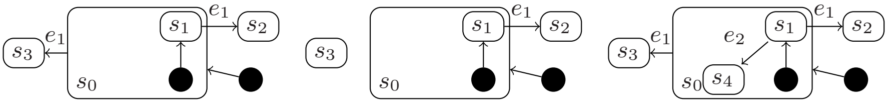


Fig. 2. State Machines

Definition 14 (Refinement). *Let SM_C and SM_A be two state machines and TS_C and TS_A their corresponding transition systems. A relation* $\mathrm{Re} \subseteq \mathcal{C}_C \times \mathcal{C}_A$ *is called a* refinement (simulation) *if:*

1. *Initial configurations are related:* $(\mathcal{S}^i_C, \mathcal{S}^i_A) \in \mathrm{Re}$.
2. *For all* $(\mathcal{S}_C, \mathcal{S}_A) \in \mathrm{Re}$ *and all* $e \in E$ *:*
   (a) *If* $\mathcal{S}_C \xrightarrow{e/\psi} \mathcal{S}'_C$, *then there exists* $\mathcal{S}'_A$ *with* $(\mathcal{S}'_C, \mathcal{S}'_A) \in \mathrm{Re}$ *and* $\mathcal{S}_A \xrightarrow{e/\psi} \mathcal{S}'_A$.
   (b) $\{s \in S_C \mid \mathsf{redef}(e, s)\} \subseteq \{s \in S_A \mid \mathsf{redef}(e, s)\}$.

If there exists a simulation relation between SM_C and SM_A, then we write $SM_C \sqsubseteq SM_A$.

The property redefinable models chaotic behavior and must be preserved by refinements. This simulation relation is transitive and implies trace inclusion.

4 Refinement Patterns

UML state machines are hierarchical machines. Transferring flat patterns to this setting requires heavy changes on the side conditions. We show this, by no means all problems, by an example. Consider the state machines in Fig. 2. Since the inner-most transition labeled e is prioritized, this transition system will never reach s_3. Thus we are allowed to *delete* the outer transition, resulting in the state machine in Fig. 2 in the middle. On the other hand, we might also add a new state and a new transition, resulting in the state machine of Fig. 2 on the right. Now we cannot remove the transition from s_0 to s_3, since it is describing a transition from the configuration $\{s_0, s_1\}$ to $\{s_3\}$, which will never be taken, and a transition $\{s_0, s_4\}$ to $\{s_3\}$, which may be taken. Therefore, removing the transition is not a transformation that preserves simulation. This demonstrates that the transformations based on flat semantics cannot be transferred to our setting without large modification.

Semantics Preserving Transformations. In this subsection we discuss semantics preserving transformations.

Definition 15. *The set of all* reachable configurations *is defined as* $creach(SM) \triangleq \{\mathcal{S} \in \mathcal{C} \mid \mathcal{S}^i \rightarrow^* \mathcal{S}\}$. *The set of all* reachable states *is defined as* $sreach(SM) \triangleq \{s \mid \exists \mathcal{S} \in creach(SM) : s \in \mathcal{S}\}$.

We may add or remove any transitions emanating a redefinable state.

Proposition 2 (Modify Transitions 1). *Let* $SM \triangleq ((S, R, \mathsf{prt}), T, \mathsf{default}, \mathsf{redef}, \mathcal{S}^i)$ *be a state machine,* s *a state and* e *an event of* SM *such that* $\mathsf{redef}(e, s)$*. Then* $((S, R, \mathsf{prt}), T' \cup T'', \mathsf{default}, \mathsf{redef}, \mathcal{S}^i) \equiv SM$ *where* $T'' \subseteq \{(s, e, \psi, s') \mid \psi \in \Psi, s' \in S\}$ *and* $T' \triangleq T \setminus \{(s, e, \psi, s') \mid \psi \in \Psi, s' \in S\}$*. Since inter-region transitions are not well formed we require:* $\forall t \in T'' : \pi_{\mathrm{sor}}(t) \sqcup \pi_{\mathrm{tar}}(t) \in R$*.*

Proof (sketch). We only need to consider configurations in which s is active. We distinguish two cases: (1) the CHAOS-rule applies to the configuration and thus no enabled transition with higher priority exists. Then in the semantics transition system all possible transitions are present, regardless whether they are also present in the syntactic representation of the state machine. (2) the STEP-rule is applied, then a transition with higher priority is fired. □

We may add or remove transitions emanating a state as long as for any reachable sub-configuration of the state there exists a fireable transition triggered by or an active substate redefinable wrt. the same event that has higher priority.

Proposition 3 (Modify Transitions 2). *Let* $SM \triangleq ((S, R, \mathsf{prt}), T, \mathsf{default}, \mathsf{redef}, \mathcal{S}^i)$ *be a state machine,* $s \in S$ *a state and* $e \in E$ *an event such that* $\forall \mathcal{S} \in \mathit{creach}(SM) : s \in \mathcal{S}$ *implies* $\mathsf{fireable}(e, \mathcal{S} \cap \upharpoonleft(s)) \neq \emptyset \vee \exists s' \in (\mathcal{S} \cap \upharpoonleft(s)) : \mathsf{redef}(e, s')$*. Then* $((S, R, \mathsf{prt}), T' \cup T'', \mathsf{default}, \mathsf{redef}, \mathcal{S}^i) \equiv SM$ *where* $T'' \subseteq \{(s, e, \psi, s') \mid \psi \in \Psi, s' \in S\}$ *and* $T' \triangleq T \setminus \{(s, e, \psi, s') \mid \psi \in \Psi, s' \in S\}$*. Again we have to require* $\forall t \in T'' : \pi_{\mathrm{sor}}(t) \sqcup \pi_{\mathrm{tar}}(t) \in R$ *in order to avoid inter-region transitions.*

Proof (sketch). Let $SM \triangleq ((S, R, \mathsf{prt}), T, \mathsf{default}, \mathsf{redef}, \mathcal{S}^i)$ be a state machine and e be an event s.t. $\forall \mathcal{S} \in \mathit{creach}(SM) : s \in \mathcal{S} \implies \mathsf{fireable}(e, \mathcal{S} \cap \upharpoonleft(s)) \neq \emptyset \vee \exists s' \in (\mathcal{S} \cap \upharpoonleft(s)) : \mathsf{redef}(e, s')$. Let $\mathcal{S} \in \mathit{creach}(SM)$ such that $s \in \mathcal{S}$. Now we have two cases: If $\exists s' \in (\mathcal{S} \cap \upharpoonleft(s)) : \mathsf{redef}(e, s')$ holds then the proof is similar to the proof of Proposition 2. If $\mathsf{fireable}(e, \mathcal{S} \cap \upharpoonleft(s)) \neq \emptyset$ then there is an enabled transition with higher priority because only states in the downset are considered. In this case the added resp. removed transitions do not contribute to the state machine's semantics. □

For any transition we may add or remove transitions with the same event, action and source. Their target state has to be the default state of the contained region.

Proposition 4 (Modify Target State). *Let* $SM \triangleq ((S, R, \mathsf{prt}), T, \mathsf{default}, \mathsf{redef}, \mathcal{S}^i)$ *be a state machine and* $t \triangleq (s, e, \psi, s')$ *a transition. Then* $((S, R, \mathsf{prt}), T' \cup T'', \mathsf{default}, \mathsf{redef}, \mathcal{S}^i) \equiv SM$ *where* $T'' \subseteq \{(s, e, \psi, s'') \mid \exists r : s'' = \mathsf{default}(r) \wedge \mathsf{prt}(r) = s'\}$ *and* $T' \triangleq T \setminus \{(s, e, \psi, s'') \mid \exists r \in R : s'' = \mathsf{default}(r) \wedge \mathsf{prt}(r) = s'\}$*.*

Proof (sketch). t as well as the transitions in T'' share the same source state s. Thus at most one of them is actually fired. Since the actions are equal there is no difference in this regard. Since the lower target states are the default states of their containing regions it follows from Definition 10 that the sames states are entered. This happens either by default, which is the third case of the definition of enter, or as the target state, which is first case. □

Any number of basic states may be added to a region of the state machine. If the states are added to the root region they are redefinable wrt. to all events. Otherwise, if the parent state of the region is not redefinable with respect to an event, then the newly added substates must not be redefinable with respect to the same event either. If the region was empty, one of the newly added states must be the default state.

Proposition 5 (Extending State Set). *Let SM* $\triangleq ((S, R, \mathsf{prt}), T, \mathsf{default}, \mathsf{redef}, \mathcal{S}^i)$ *be a state machine and* $r \in R$ *and* S' *such that* $S \subset S'$*. For any state* $s \in S'$ *and event* $e \in E$ *define* $\mathsf{redef}'(e, s) \triangleq \mathsf{redef}(e, s)$ *if* $s \in S$, $\mathsf{redef}'(e, s) \triangleq \top$ *if* $r = \epsilon$ *and* $\mathsf{redef}'(e, s) \triangleq \mathsf{redef}(e, \mathsf{prt}(r))$ *otherwise. For any vertex* $v \in S' \cup R$ *define* $\mathsf{prt}'(v) \triangleq \mathsf{prt}(v)$ *if* $v \in S \cup R$ *and otherwise* $\mathsf{prt}'(v) \triangleq r$*. For any region* $r' \in R$ *define* $\mathsf{default}'(r') \triangleq \mathsf{default}(r')$ *if* $r' \neq r$ *or* $r \in dom(\mathsf{default})$*, and* $\mathsf{default}'(r) \triangleq s'$ *for some state* $s' \in S' \backslash S$*. Define* $\mathcal{S}^{i'} \triangleq \mathcal{S}^i$ *if* $\mathsf{prt}(r) \notin \mathcal{S}^i$ *and* $\mathcal{S}^{i'} \triangleq \mathcal{S}^i \cup \{\mathsf{default}'(r)\}$ *otherwise . Then SM* $\equiv ((S', R, \mathsf{prt}'), T, \mathsf{default}', \mathsf{redef}', \mathcal{S}^{i'})$.

We may remove any non reachable basic states and transitions emanating these states. Functions parent, default and redefinable are restricted to the new set of states.

Proposition 6 (Remove Non Reachable Basic States). *Let SM* $\triangleq ((S, R, \mathsf{prt}), T, \mathsf{default}, \mathsf{redef}, \mathcal{S}^i)$ *be a state machine,* $S' \triangleq (S \setminus \mathrm{sreach}(SM)) \cap \{s \in S \mid \forall r : \mathsf{prt}(r) \neq s\}$ *and* $T' \triangleq \{t \in T \mid \pi_{\mathrm{sor}}(t) \in S \setminus S'\}$*. Let* prt' *equal* prt, $\mathsf{default}'$ *equal* $\mathsf{default}$ *and* redef' *equal* redef *on domain* $(S \setminus S') \cup R$*. Then SM* $\equiv ((S \setminus S', R, \mathsf{prt}'), T', \mathsf{default}', \mathsf{redef}', \mathcal{S}^i)$.

A new region may be added to any state.

Proposition 7 (Add Regions). *Let SM* $\triangleq ((S, R, \mathsf{prt}), T, \mathsf{default}, \mathsf{redef}, \mathcal{S}^i)$ *be a state machine and* $s \in S$*. Then SM* $\equiv ((S, R', \mathsf{prt}'), T, \mathsf{default}, \mathsf{redef}, \mathcal{S}^i)$ *where* $R \subset R'$*, and for any vertex* $v \in S \cup R'$ *define* $\mathsf{prt}'(v) \triangleq s$ *if* $v \in R' \setminus R$ *and* $\mathsf{prt}'(v) \triangleq \mathsf{prt}(v)$ *otherwise.*

All empty regions, except the root region, may be removed.

Proposition 8 (Remove Empty Regions). *Let SM* $\triangleq ((S, R, \mathsf{prt}), T, \mathsf{default}, \mathsf{redef}, \mathcal{S}^i)$ *be a state machine and* $R' \triangleq R \setminus \{r \in R \mid \forall s \in S : \mathsf{prt}(s) \neq r \wedge r \neq \epsilon\}$*. Let* prt' *equal* prt *on domain* $S \cup R'$*. Then SM* $\equiv ((S, R', \mathsf{prt}'), T, \mathsf{default}, \mathsf{redef}, \mathcal{S}^i)$.

Note that we can add resp. remove a non reachable state containing regions and other states by multiple successive application of Prop. 7 and Prop. 5 resp. Prop. 8 and Prop. 6.

Definition 16 (State Hierarchy Epimorphism). *Let* (S', R', prt') *and* (S, R, prt) *be two state hierarchies. A function* $\alpha : (S', R', \mathsf{prt}') \to (S, R, \mathsf{prt})$ *is called a* state hierarchy epimorphism, *if:* $\alpha : S \to S'$ *is total and onto,* $\alpha : R \to R'$ *is total and onto,* $\alpha(\epsilon) = \epsilon$*, and* $\alpha(\mathsf{prt}(v)) = \mathsf{prt}'(\alpha(v))$*. To avoid the duplication of actions, we require that two regions must not be mapped into the same parent state, i.e. we do not copy regions without their containing state:*
$\forall r, r' \in R' \setminus \{\epsilon\} : \mathsf{prt}'(r) = \mathsf{prt}'(r') \Rightarrow \alpha(r) \neq \alpha(r') \vee r = r'$

We may duplicate states together with their incoming and outgoing transitions.

Proposition 9 (Duplicating States and Transitions). *Let* $SM \triangleq ((S, R, \mathsf{prt}), T, \mathsf{default}, \mathsf{redef}, \mathcal{S}^i)$ *be a state machine,* (S', R', prt') *a state hierarchy such that* $\alpha : (S', R', \mathsf{prt}') \to (S, R, \mathsf{prt})$ *is a state hierarchy epimorphism. Define*

1. $T' = \{(s, e, \psi, s') \mid (\alpha(s), e, \psi, \alpha(s')) \in T \wedge \alpha(\mathsf{lcr}(\{s, s'\})) = \mathsf{lcr}(\{\alpha(s), \alpha(s')\})\}$
2. $\forall r' \in R' : \mathsf{default}(\alpha(r')) = \alpha(\mathsf{default}'(r'))$
3. $\forall s \in S' : \mathsf{redef}'(e, s) = \mathsf{redef}(e, \alpha(s))$
4. $\mathcal{S}^{i\prime} = \mathsf{enter}'(\epsilon, \mathsf{default}'(\epsilon))$

Then $((S', R', \mathsf{prt}'), T', \mathsf{default}', \mathsf{redef}', \mathcal{S}^{i\prime}) \equiv SM$*.*

Example 3. Consider the state machine in Fig. 3 on the left. Duplicating all states and regions below the root region yields the state machine depicted in Fig. 3 on the right. The corresponding state hierarchy epimorphism maps states identified by a letter and a number to the state identified by the letter. Since $\mathsf{default}'$ has to be a function, either E_1 or E_2 can be the default state in the root region. One of them has to be the default state, because otherwise $\mathsf{default}(\alpha(\epsilon)) = \mathsf{default}(\epsilon) = E$ and $\mathsf{default}'(r)$ is undefined, which contradicts condition 2.

A transition t' from A_1 to B_2 labeled with e/ψ_1 must not be added, even though $\alpha(A1) = A$, $\alpha(B_2) = B$ and there is a transition from A to B labeled with e/ψ_1: When the states $\{A, C, E\}$ are active and event e is dispatched, the state machine in Fig. 3 on the left reacts with ψ_1, ψ_2. When the states $\{A_1, C_1, E_1\}$ are active, the state machine in Fig. 3 on the right reacts with ψ_1 only. Therefore Fig. 3 on the right cannot refine Fig. 3 on the left with t' added. Adding t' is prevented by $\alpha(\mathsf{lcr}(\{s, s'\})) = \mathsf{lcr}(\{\alpha(s), \alpha(s')\})$.

Assume that the topmost object in the hierarchy to be duplicated is a region which contains two states A and B and there is a transition from A to B labeled e/ψ_1. Now duplicating the region duplicates A and B to A_1, A_2, B_1, B_2. Similar as above $\alpha(A_1) = A$, $\alpha(B_2) = B$ holds and the transition t'' from A_1 to B_2 labeled with e/ψ_1 must not be added, too, because the transition leaves one region and enters the other region without leaving or entering the state containing both regions. This is not a valid transition according to the UML semantics. Adding t'' is prevented by the constraint $\alpha(\mathsf{lcr}(\{s, s'\})) = \mathsf{lcr}(\{\alpha(s), \alpha(s')\})$.

Reducing Nondeterminism. So far we only discussed patterns with simulation equivalence. We now present patterns that reduce nondeterminism and thereby remove behavior from the system.

We may remove the redefinable property from states as long as Property (1) of redefinable presented right after Example 1 is preserved.

Proposition 10 (Remove Redefinable). *Let* $SM \triangleq ((S, R, \mathsf{prt}), T, \mathsf{default}, \mathsf{redef}, \mathcal{S}^i)$ *be a state machine. Let* redef' *be such that* $\forall e \in E : \forall s \in S : \mathsf{redef}'(e, s) \implies \mathsf{redef}(e, s) \wedge \forall s' \in \downarrow(s) \cap S : \mathsf{redef}'(e, s')$*. Then* $((S, R, \mathsf{prt}), T, \mathsf{default}, \mathsf{redef}', \mathcal{S}^i) \sqsubseteq SM$*.*

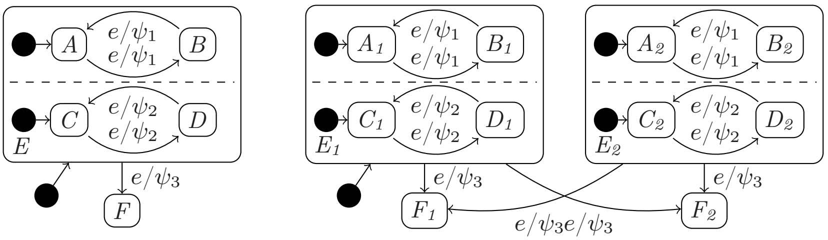


Fig. 3. A state machine before(left) and after(right) state duplication

Proof (sketch). Any reachable configuration $\mathcal{S}$ of $((S, R, \mathsf{prt}), T, \mathsf{default}, \mathsf{redef}', \mathcal{S}^i)$ is a reachable configuration of SM. From the definition of redef' follows that $\{s \in \mathcal{S} \mid \mathsf{redef}'(e, s)\} \subseteq \{s \in \mathcal{S} \mid \mathsf{redef}(e, s)\}$ with equality as simulation. □

If a state is not redefinable wrt. an event and has more than one outgoing transition triggered by that event, one of the transitions can be removed if all sets of fireable transitions containing the transition to be removed are not fireable after removal.

Proposition 11 (Remove Transitions). *Let* $SM_A \triangleq ((S, R, \mathsf{prt}), T, \mathsf{default}, \mathsf{redef}, \mathcal{S}^i)$*,* $e \in E$*,* $s \in S$ *such that not* $\mathsf{redef}(e, s)$ *and* $T' \triangleq \{t \in T \mid \pi_{\mathrm{sor}}(t) = s \wedge \pi_{\mathrm{ev}}(t) = e\}$*,* $t \in T'$ *and* $SM_C \triangleq ((S, R, \mathsf{prt}), T \setminus \{t\}, \mathsf{default}, \mathsf{redef}, \mathcal{S}^i)$*. If* $|T'| > 1$ *and* $\forall \mathcal{S}_A \in \mathcal{C}_A : \forall T'' \in \mathsf{fireable}_{SM_A}(\pi_{\mathrm{ev}}(t), \mathcal{S}_A) : t \in T'' \Rightarrow T'' \setminus \{t\} \notin \mathsf{fireable}_{SM_C}(\pi_{\mathrm{ev}}(t), \mathcal{S}_A)$*. Then* $SM_C \sqsubseteq SM_A$*.*

Proof (sketch). Let t be the deleted transition. Because the states and redef remain unaffected we can use equality as simulation relation, i.e. $(\mathcal{S}_C, \mathcal{S}_A) \in \mathrm{Re} \Leftrightarrow \mathcal{S}_C = \mathcal{S}_A$. Let $(\mathcal{S}_C, \mathcal{S}_A) \in \mathrm{Re}$. Let $e \in E$ and $\mathcal{S}_C \xrightarrow{e/\psi} \mathcal{S}'_C$ such that $e = \pi_{\mathrm{ev}}(t)$ and $\pi_{\mathrm{sor}}(t) \in \mathcal{S}_C$, otherwise we have $\mathcal{S}_A \xrightarrow{e/\psi} \mathcal{S}'_A$ where $\mathcal{S}'_C = \mathcal{S}'_A$. Let T'' be the set of transitions that is fired when commencing step $\mathcal{S}_C \xrightarrow{e/\psi} \mathcal{S}'_C$. We have $|T''| > 0$ since $|\{t' \in T \mid \pi_{\mathrm{sor}}(t) = \pi_{\mathrm{sor}}(t') \wedge \pi_{\mathrm{ev}}(t) = \pi_{\mathrm{ev}}(t')\}| > 1$. We show that $T'' \in \mathsf{fireable}_{SM_A}(\pi_{\mathrm{ev}}(t), \mathcal{S}_A)$ and thus $\mathcal{S}_A \xrightarrow{e/\psi} \mathcal{S}'_A$ where $\mathcal{S}'_C = \mathcal{S}'_A$. Suppose $T'' \notin \mathsf{fireable}_{SM_A}(\pi_{\mathrm{ev}}(t), \mathcal{S}_A)$ then $T'' \cup \{t\} \in \mathsf{fireable}_{SM_A}(\pi_{\mathrm{ev}}(t), \mathcal{S}_A)$ otherwise we have $T'' \notin \mathsf{fireable}_{SM_C}(\pi_{\mathrm{ev}}(t), \mathcal{S}_C)$. This is a contradiction to $\forall \mathcal{S}_A \in C_A : \forall T'' \in \mathsf{fireable}_{SM_A}(\pi_{\mathrm{ev}}(t), \mathcal{S}_A) : t \in T'' \Rightarrow T'' \setminus \{t\} \notin \mathsf{fireable}_{SM_C}(\pi_{\mathrm{ev}}(t), \mathcal{S}_A)$. □

Refinement relations are based on the semantics and not in the syntactic model. In flat state machines a step in the semantics corresponds to the firing of at most one transition. Then deletion of a transition in the model removes all corresponding steps in the semantics, and the side condition is always satisfied. The side condition is caused by hierarchy and orthogonal regions, where the semantic steps are composed of more than one syntactic transition.

Example 4. Consider the state machine in Figure 4 with active states $\{A, C, E\}$. On event e the machine can evolve to $\{B, D, E\}$ with actions ψ_1, ψ_2 and to $\{F\}$

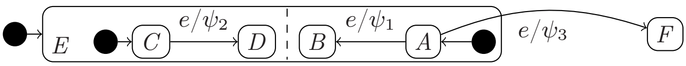


Fig. 4. Removal of conflicting transitions

with action ψ_3. When the transition from A to B is removed, the following reactions are possible: With action ψ_2 to $\{A, D, E\}$ and with action ψ_3 to $\{F\}$. Removing the transition does not *remove* behavior but it *changes* it. The set $\{(A, e, \psi_1, B), (C, e, \psi_2, D)\}$ is a set of fireable transitions. After deleting transition (C, e, ψ_2, D) $\{(A, e, \psi_1, B)\}$ is a set of fireable transitions. Therefore deleting the transition is not a refinement according to Prop. 11.

5 Conclusions, Related Work, and Future Work

We have studied refinement patterns for hierarchical UML state machines. We believe that our semantics is closest to the semantics defined in the UML standard [5] when compared to work cited below. We also believe, that we cover the richest subset of the UML state machine language. We are aware that the patterns presented in this paper are not complete but that is not the goal of the paper. Rather than providing more, probably more complex and confusing, patterns we selected a basic subset of transformations. In any case changes of the model, not covered by the patterns presented, can be checked with standard automated refinement techniques on the semantic transition systems. Therefore we believe that the transformations presented are valuable basis in a practical setting. The question whether there are additional simple and useful patterns is future work.

In order to facilitate the development of state machines, we have introduced a new *stereotype* «redef E», which expresses that the annotated state is underspecified with respect to the reaction to events of E. Without this stereotype, UML state machines drop events to which no reaction is specified.

The most comprehensive discussion of refinement of automata in an object-oriented setting can be found in Bernhard Rumpe's doctoral thesis [14]. He presents a formalism in which underspecification of event handling allows arbitrary behavior. Behavior is constrained if transitions are defined in the state machines that handle the given event. This is in contrast to UML state machines, where an event that is not handled by a transition is removed from the event pool without a state change or a reaction.

From a methodical point of view, explicit underspecification is an advantage, because state machines may drop events if they cannot be handled. Rumpe [14] leaves underspecification implicit, where it is not apparent whether the development of the system has been "finished."

Sun Meng et al. [15] propose refinement patterns similar to ours. Their semantics is based on Lattela et.al. [7], where the priority relation is inverted to ours: Sun Meng delegates events "downwards" to substates whereas we delegate them upwards towards the superstates, as required in [5]. Another difference stems from a different interpretation of the *stutter rule*: If an event cannot be handled by the state machine, Meng defers the event whereas we drop the event.

The refinement patterns for μ-charts of Reeve and Reeves [19] do not carry to our setting, either. μ-charts have parallel regions, but they have no hierarchy and, consequently, no interlevel transitions. The theory deals with *feedback* of events, i.e., events generated in a transition are feed back to the state machine. This does not happen in UML state machines. As Rumpe, Reeve and Reeves default to chaotic behavior and restrict by adding transitions, whereas we specify chaotic behavior explicitly.

The main results are: Some transformations proposed by Rumpe do not carry over to Sun Meng's or our setting, since the language features do not carry over. Other patterns appear to be quite general, but have different side conditions under which the transformation is a refinement. Compare, for instance the rule for adding transitions, Proposition 2 in this paper, Proposition 6.6 in [14] and Law 5 in [15]. In [14], a transition may be added, if no other transition leaving the same source state exists, because he considers that state under-specified, and any behavior is allowed if nothing is specified. In [15], a transition may be added under a the same condition, but there, lack of specified behavior causes stuttering. Finally, in our case the side conditions are more complex, since the state machine drops events if nothing is specified. This cannot be refined freely, since it might change subsequent behavior, or generate unexpected actions. Therefore, we mark states with redefinable to specify that the state is underspecified.

Many features of the UML state machine language have not been considered here. Some of those features can be covered easily, like fork and join states, entry points, exit points, and junction. We did not present our results for these features for lack of space. Local and internal transitions are also easy to include, because they are semantically similar. Syntax and semantics can be found in [12], and the patterns need only marginal changes. Other language features are harder challenges, e.g., history pseudo states and final states.

UML allows to associate actions to states, which are executed on entering or leaving a state (this can be simulated in our setting) and also while the state is active. The latter may be investigated in the future.

Also, event deferral is not considered. If an event is marked as "deferrable" and no transition is triggered by this event, then it will not be considered for selection, effectively disabling the DISCARD rule. The resulting patterns are similar to [15].

Object states, variable states, and actions that transform these are not considered. Our requirement of *equality* of actions implied in Def. 14 already prepares for handling these. The refined system may use any sequence of actions that is a data-refinement of the original action. This is outside the scope of this paper and treated exhaustively in, e.g., de Roever and Engelhardt [16].

References

1. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3), 231–274 (1987)
2. Harel, D., Gery, E.: Executable object modeling with Statecharts. Computer 30(7), 31–42 (1997)

3. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc., New York (1994)
4. Maraninchi, F., Rémond, Y.: Argos: an automaton-based synchronous language. Comp. Lang. 27(1/3), 61–92 (2001)
5. OMG: UML 2.1.2 Superstructure Specification (November 2007), `http://www.omg.org/cgi-bin/docs/formal/2007-11-02.pdf`
6. Fecher, H., Schönborn, J., Kyas, M., de Roever, W.P.: 29 new unclarities in the semantics of UML 2.0 state machines. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 52–65. Springer, Heidelberg (2005)
7. Latella, D., Majzik, I., Massink, M.: Towards a formal operational semantics of UML Statechart diagrams. In: Ciancarini, P., Gorrieri, R. (eds.) FMOODS, pp. 331–347. Kluwer Academic Publishers, Dordrecht (1999)
8. von der Beeck, M.: A structured operational semantics for UML-statecharts. Software and Systems Modeling 1(2), 130–141 (2002)
9. Bianco, V.D., Lavazza, L., Mauri, M.: A formalization of UML Statecharts for real-time software modeling. In: The Sixth Biennal World Conference of Integrated Design Process Technology, IDPT 2002 (2002)
10. Jürjens, J.: Formal semantics of interacting uml subsystems. In: Jacobs, B., Rensink, A. (eds.) FMOODS, pp. 29–44. Kluwer Academic Publishers, Dordrecht (2002)
11. Börger, E., Cavarra, A., Riccobene, E.: Modeling the meaning of transitions from and to concurrent states in UML state machines. In: SAC, pp. 1086–1091. ACM Press, New York (2003)
12. Fecher, H., Schönborn, J.: UML 2.0 state machines: Complete formal semantics via core state machine. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 244–260. Springer, Heidelberg (2007)
13. Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981)
14. Rumpe, B.: Formale Methodik des Entwurfs verteilter objektorientierter Systeme. Herbert Utz Verlag Wissenschaft (1997)
15. Meng, S., Naixiao, Z., Barbosa, L.S.: On semantics and refinement of UML Statecharts: A coalgebraic view. In: SEFM, pp. 164–173. IEEE, Los Alamitos (2004)
16. de Roever, W.P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. Cambridge Tracts in Theoretical Computer Science, vol. 47. Cambridge University Press, Cambridge (1998)
17. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order, 2nd edn. Cambridge University Press, Cambridge (2002)
18. van Glabbeek, R.J.: The linear time-branching time spectrum (extended abstract). In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458, pp. 278–297. Springer, Heidelberg (1990)
19. Reeve, G., Reeves, S.: Logic and refinement for charts. In: Estivill-Castro, V., Dobbie, G. (eds.) ACSC. CRPIT, vol. 48, pp. 13–23. Australian Computer Society (2006)

Specification and Validation of Behavioural Protocols in the rCOS Modeler

Zhenbang Chen[1,2], Charles Morisset[1], and Volker Stolz[1]

[1] United Nations University
Institute for Software Technology
P.O.Box 3058, Macau SAR
{zbchen,cm,vs}@iist.unu.edu
[2] National Laboratory for Parallel and Distributed Processing
Changsha, China

Abstract. The rCOS modeler implements the requirements modelling phase of a model driven component-based software engineering process. Components are specified in rCOS, a relational calculus for Refinement of Component and Object Systems. As an aid to the software engineer, the modeler helps to separate the different concerns by creating different artifacts in the UML model: use cases define a scenario through a sequence diagram, and methods are given as guarded designs in rCOS. rCOS interface contracts are specified through state machines with modelling variables. Messages and transitions in the diagrams are labelled with method invocations.

The modeler checks the consistency of those artifacts through the process algebra CSP and the model checker FDR2: a scenario must follow a contract, and an implementation must not deadlock when following the contract. We illustrate the translation and validation with a case study.

1 Introduction

Software engineering is becoming more complex due to the increasing size and complexity of software products. *Separation of concerns* is an effective means to tackle modelling of complex system. The quality of a system can be improved by applying formal techniques in different development stages.

In our previous work, we introduced the notion of an *integrated specification* that derives a specification of a component-based system from a UML-like model for a use case [2]. A use case defines a syntactic *interface* (the provided methods) and controller class implementing this interface plus its referenced data structures, a system *sequence diagram* involving only a single actor and the component that describes the external behaviour, and a *state machine* describing the internal behaviour of the component. In the sequence diagram and the state machine, guarded transitions are labelled with methods from the interface.

Apart from the usual notion of well-definedness (also called *static consistency*) for an object-oriented design, we require the *dynamic consistency* of the specification: the state machine must accept all interaction sequences

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 387–401, 2010.

described by the system sequence diagram, and any implementation of the interface must not deadlock if invoked according to the *protocol* given through the state machine.

We have implemented the requirements modelling stage of a use case-driven model-based component development process following the rCOS methodology in the rCOS modeler. An rCOS model is a UML model extended through the rCOS UML profile [5]. The tool supports static checking of the dynamic consistency of the model through semi-automated translation into the process algebra CSP and the model checker FDR2 [19,6].

We define an automated abstraction into 'flat' rCOS that only uses primitive types. The abstraction is further parametrized according to criteria specified by the user, e.g. to hide method- (and thus message-) arguments/return parameters. This flat representation is then translated into CSP, which we use to check that the generated rCOS design follows the protocol by checking the deadlock freedom of parallel composition of the generated CSP processes.

The paper is organized as follows: Section 2 presents our approach of separation of concerns including the underlying theory, the used modelling artifacts, and a brief explanation of integrated specification and checking; we explain the dynamic consistency checking in Section 3, which contains the translation from the rCOS model to CSP and the abstraction method used in the translation; finally, Section 4 concludes.

Related work. Olderog *et al.* present CSP-OZ, a formal method combining CSP with the specification Object-Z, with UML modelling and Java implementations [12]. They use a UML profile to annotate the model with additional data. Model properties can then be verified on the CSP, and their notion of contracts of orderings between method invocations through JML and $\text{CSP}_{\text{jassda}}$ can be enforced on Java programs at runtime. Their tool `Syspect` is also built on the Eclipse Rich Client Platform.

Executable UML [11] introduces a UML profile that gives a suitable semantics for direct execution to a subset of UML. As such, it focuses on execution and not formal verification. Use cases and state diagrams are used, procedures are specified in an action language.

The practicability of generating a PROMELA specification for the Spin model checker from rCOS has been investigated in [22]. The semantics of the rCOS (PROMELA) specification is derived from executing the `main` method. The model is executed by the model checker for a bounded number of objects and invariants are checked.

Snook and Butler [20] use B as the underlying theory during the modeling and design process using UML. The class diagrams and state diagrams in UML can be translated to a B description, including the function specifications and guards for operations. They also use a state variable in B to represent transitions during the translation of a state diagram. The refinement notion in B supports refinement between different UML models in the development stages. Ng and Butler discuss a similar translation of UML state machines to CSP in [14,13].

Our concern in this paper is checking the consistency of multi-view specifications. The translation to CSP also extends to the verification of component composition in the rCOS theory (see [1]).

2 Separation of Concerns

The rCOS language is based on UTP, Unifying Theories of Programming [8], and is object-oriented. We give here a brief description of its main features. We refer to [4] for further details.

Method. A method $m \in \mathit{Meth}$ is a tuple $m = (\mathit{Name}, \mathit{in}, \mathit{out}, g, d)$ where *Name* is the name of the method, *in* (resp. *out*) is a set of input parameters together with their type, $g \in \mathbb{G}$ is the guard and $d \in \mathbb{D}$ is the design. A guard is a boolean expression, which does not contain any primed variable from the post-state, and cannot refer to parameters of the method. A design could be

- a pre/post-condition pair $[p \vdash R]$, where p and R are predicates over the observables,
- a conditional statement $d_1 \lhd e \rhd d_2$, where d_1 and d_2 are designs and e is a boolean expression,
- a sequence $d_1; d_2$, where d_1 and d_2 are designs,
- a loop $\star(e, d_1)$, where d_1 is a design and e is a boolean expression or
- an atomic command, such as an assignment, a variable declaration, a method call, SKIP or CHAOS.

Interfaces. An *interface* $I \in \mathbb{I}$ is a tuple $I = (\mathit{FDec}, \mathit{MDec})$ where *FDec* is the *fields declaration section* and *MDec* the *method declaration section*. Each signature is of the form $(m, \mathit{in}, \mathit{out})$, where *in* are input parameters and *out* are the output parameters.

Class. A class $c \in \mathbb{C}$ is a tuple $c = (\mathit{FDec}, \mathit{MSpec})$ where *FDec* is a set of fields and $\mathit{MSpec} \subseteq \mathit{Meth}$ a set of methods. We assume that we can project onto the public and private attributes (or methods) of interfaces (and classes respectively) through $I.\mathit{FDec}_{pub}$ and $I.\mathit{FDec}_{priv}$.

Contracts of Interfaces. A *contract* $\mathit{Ctr} = (I, \mathit{Init}, \mathit{MSpec}, \mathit{Prot})$ specifies

- the allowable initial states by the initial condition *Init*, a predicate over the attributes in I.
- the synchronization condition g on each declared method and the functionality of the method by the specification function $\mathit{MSpec} : I.\mathit{MDec} \rightarrow (\mathbb{G} \times \mathbb{D})$ that assigns each method defined in I to a guarded design $g\&D$.
- *Prot* is called the *protocol* and is a set of sequences of call events; each is of the form $?op_1(x_1), \ldots, ?op_k(x_k)$. A protocol can be specified through many different means, here we will consider protocols generated from sequence diagrams and state machines.

UML-based Requirement Specification

The models are defined in the Unified Modeling language (UML), and we apply Model Driven Development and Architecture (MDD/MDA) techniques [16].

For the modelling part, we use UML models and a UML profile to tie together the necessary information, e.g. by assigning which specification belongs to a use case. The specifications are bundled in packages and tagged with stereotypes from the profile to mark them as belonging to the rCOS modeling domain. The profile is documented in [5].

The advantage of using UML is three-fold: firstly, we can provide the familiar modelling notations for the system developer, yet augmented with a rigorous underlying mathematical semantics; secondly, we can reuse the UML meta-model by using a profile to get an rCOS meta-model, because rCOS intentionally overlaps with UML; lastly, there are numerous tools and methodologies for UML, and UML models are the de facto standard models that they create or exchange, so we can harness their powers to enhance the rCOS tool, e.g. we support importing from the UML tool MagicDraw [15], and also re-used an existing graphical UML modeler, saving us development effort.

2.1 Example

We use one use case (called ProcessSale) of our recent case study [3] as the example, which is a trading system based on Larman's textbook [9], originally called the *Point of Sale* (POS) system. The trading system records sales, handles both cash payments and card payments as well as inventory management. The system includes hardware components such as computers, bar code scanners, card readers, printers, and software to run the system. The *normal courses* of interactions in the *ProcessSale* use case are informally described:

1. The *cashier* sets *checkout mode* to express check out or normal check out.
2. When a *customer* comes to the *checkout* with their *items* to purchase, the *cashier* indicates the system to handle a new *sale*.
3. The *cashier* enters each item, either by typing or scanning in the *bar code*; if there is more than one of the same item, the cashier can enter the *quantity*. The system records each item and its quantity and calculates the subtotal. When the cash desk is operating in *express mode*, only a predefined maximum number of items can be entered.
4. When there are no more items, the *cashier* indicates to the system *end of entry*. The *total* of the sale is calculated.
5. The customer can pay by cash or credit card. If by cash, the amount received is entered. In express mode, only cash payment is allowed. After payment, the inventory of the store is updated and the completed sale is logged.

There are *exceptional* or *alternative courses* of interactions, e.g., the entered bar code is not known in the system. At the requirements level, we capture these exceptional conditions as preconditions.

2.2 Integrated Specification

In the *requirements modelling* stage of software development, each use case is modelled as a component, with the specification of the contract of its provided interface containing the interaction protocol, the reactive behaviour, and the functionalities the methods provided in the interface. The main advantage of the rCOS methodology is that we can assure consistency of the multi-view specifications [10], for example by checking trace equivalence or deadlock freedom of the diagrams. We generate appropriate CSP specifications [3] for the FDR2 model checker [7,6]. While looking at state machines for verification, combining UML and CSP is certainly not new, cf. [14], but we hope to make state diagrams more prominent in a formal development process, and we also consider guarded designs. The overview of the integrated specification and checking is shown in Fig. 1: the interface serves as the signature of a contract that contains a sequence diagram for the scenario and the state machine for the protocol. A component can implement a contract, and a component can be refined. We later discuss the different operations like generating CSP from the different artifacts, or integrating the contract with a functionality specification (a class containing the business logic) into a guarded design that implements the contract. Together with appropriate helper functions for abstractions, we can then turn an implementation into a 'flat' rCOS program. We can then use again CSP to verify that the implementation actually follows a contract. This is interesting for implementations coming from third parties that claim to respect a particular contract.

The component in the contract box aggregates the relevant data like objects, classes, and their associations taking part in the use case. The data types and classes are modelled as a class diagram that is derived from the problem description. We borrow the term "conceptual" class diagram from Larman [9] to indicate that at this stage, we do not assign visibility to the attributes and assume that they are all public. Also, there are initially no methods except for the

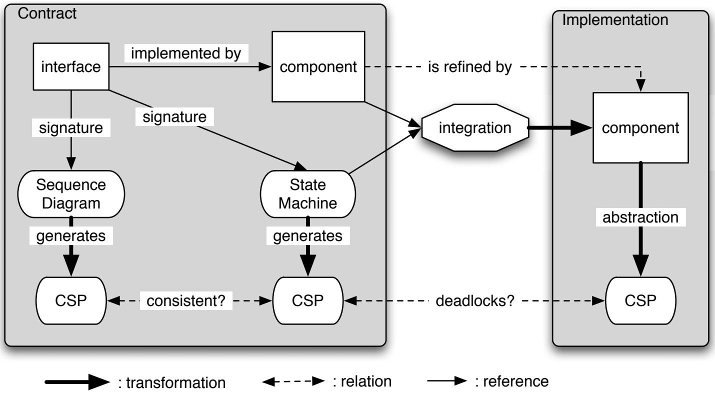


Fig. 1. Overview of specification and checking

controller class implementing the provided interface, and each method of the controller class can be specified with a guard and the function specification. The function specification of the *enterItem* method in the control class *CashDesk* for the *CashDeskIF* contract of the *ProcessSale* use case is as follows.

```
public enterItem(Barcode code, int qty ; ) {
  VAR LineItem item ;
  [ pre  : store.catalog.find(c) != null ,
    post : line' = LineItem.new(c, q) ;
        line.subtotal' = store.catalog.find(c).price * q;
            sale.lines.add(line)]  ;
  [ ⊢ itemCounter' = itemCounter + 1 ]
}
```

In rCOS, the notation $[p \vdash R_1; R_2]$ stands for $[p \vdash R_1]; [true \vdash R_2]$ and that for each post-condition, there is an implicit statement leaving all the variables non concerned by the post-condition unchanged.

For different concerns, the *sequence* and *state diagrams* illustrate the interaction of the user with the system, which will have to conform to the protocol in the component contract. We allow only a limited use of the UML sequence diagram (collaboration) facility: there is only one actor (the user) and one process (the system). Messages only flow from the user to the system and represent *invocations* of methods in the component interface. We allow the usual control structures such as iteration and conditional branches in the sequence diagram. These have controlling expressions in the form of boolean queries or counters.

While the sequence diagram describes the possible interactions with the system the user can have, the *state machine* describes the *contract* of the provided interface. Edges in the UML state machine are labelled with an operation of the interface, which may have the form g & $op(\bar{x}; \bar{y})$, indicating that this edge may be

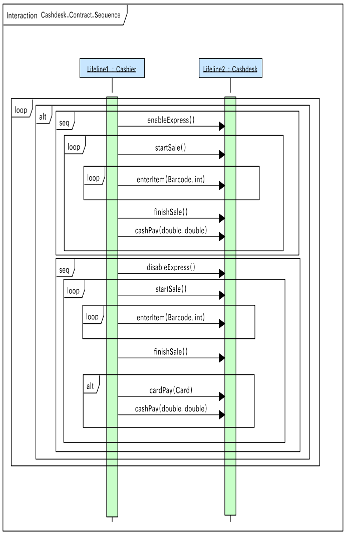


Fig. 2. Sequence Diagram

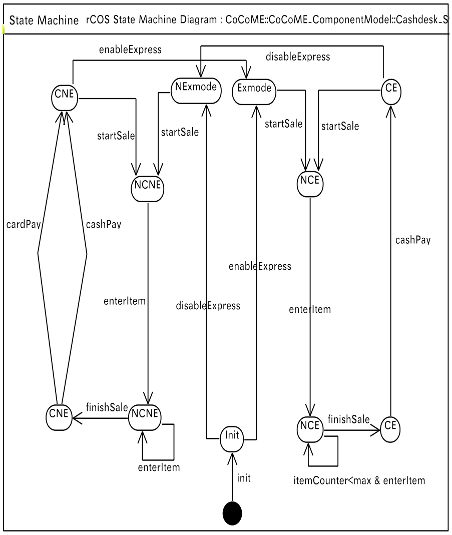


Fig. 3. State Diagram

triggered by an invocation of method *op* iff the guarding expression g is evaluated to be true. We allow nondeterminism by having multiple outgoing edges from the same state, each labelled with the same method and potentially overlapping guards for specification purposes, but point it out to the user as potentially undesired in the light of a future implementation of the protocol. Choice nodes with a boolean expression allow if-then-else constructs, or non-deterministic internal choice (not present in example). The protocol of the *CashDeskIF* contract is modelled by the sequence diagram in Fig. 2 and the dynamic flow of control by the state diagram in Fig. 3.

Naturally, there is a close relation between the *trace languages* over the method calls induced by the sequence diagram and the state diagram: the state machine must at least accept the runs of the sequence diagram. Conversely, the problem description should specify if the state machine is allowed to offer *more behaviour* than the sequence diagram.

3 Verification

For the translation of UML state machines into CSP, we limit ourselves to a subset of the features available in UML. Some of the limitations are arbitrary and based on the practical diagrams that we use in the case study. Some obvious extensions and shorthands are left open as future work.

We only allow a single initial state and plain states, connected by transitions. State labels do not carry any semantics and are just informative; they may be used e.g. to generate labels in subsequent stages.

Given a contract *Ctr*, transitions between states are labelled with an operation $op(\bar{x};\bar{y}) \in I.MDec$ from the associated interface I of the contract that have a 'flat' guarded design $g_{Ctr_{op}}$ & $D_{Ctr_{op}}$. In an integrated specification, the operation *op* is mapped to a guarded design $g_{C_{op}}$ & $D_{C_{op}}$ in a class C. The guarding expression $g_{Ctr_{op}}$ may only refer to private attributes of the interface (modelling variables), and g_{op} only to members of the class. Missing guards are assumed to be *true*. We do not use UML's effects to designate an activity that occurs when a transition fires, but will instead use the method specification from the contract.

For the translation to CSP, the rCOS guarded designs must be mapped into CSP. As this is generally hard to automate because some abstraction needs to be applied, we proposed that this step is done by a verification engineer [10].

We simplify and automate things as much as possible in the tool: for example, constructs over integers (or sets thereof) can usually be trivially abstracted into CSP. We expect that modelers use mostly primitive types in guards of state machines and sequence diagrams. Also the guarded design derived from an integrated specification uses integers to identify the current and successor state.

Objects and object references of course pose a problem, since CSP has no concept of object orientation. Thus objects and any use of navigation in complex expressions must be broken down. Since we can handle 'flat' rCOS, that only uses primitive types, more easily as described above, we solve the problem by allowing the verification engineer to define an *abstraction function*

$\alpha : rCOS \to rCOS$. α should preserve the refinement relation, that is formally in rCOS, $\forall m \in rCOS : m \sqsubseteq \alpha(m)$. That way, we translate an object oriented rCOS specification into CSP by going through an intermediate step where the verification engineer applies the abstraction function (in this case from object-oriented to 'flat' rCOS), from which we then generate the CSP. We intend to provide a library of abstractions, for example, only considering object identifiers (again in the form of integer values) for objects, and for common types like sets and bags. For the latter, there can be different granularity, like only considering if such a structure is empty or if it *may be* full in a three-valued abstraction. Another simple abstraction is hiding variables. Since the result has to be another valid rCOS specification, all expressions referring to the now hidden variables have to be properly abstracted as well. Currently it is the responsibility of the user to consider the implications of his chosen abstraction with regard to soundness and completeness of any property of the abstracted model, and the impact on the size of the state space of the model.

Let $\mathbb{C}\textsc{sp}$ denote a syntactically and statically valid CSP specification consisting of a set of process definitions. We consider now contracts in $\mathbb{C}tr_{SM}$, that are contracts for which the protocol is defined by a state machine $sm = \langle i : \mathbb{V}, S : 2^{\mathbb{V}}, trans : 2^{(\mathbb{V} \times MName \times \mathbb{V})}\rangle$, where $trans \subseteq (S \times MName \times S)$ is the relation defining transitions between states and operations of the interface.

We define the translation function for a contract $csp : \mathbb{C}tr \to \mathbb{C}\textsc{sp}$ by following the outgoing transitions from the initial state of the state machine (denoted in the following by $Ctr.sm$). This also eliminates unreachable parts. Whether we chose a breadth-first or depth-first strategy does not matter. $csp_{sm} : \mathbb{V} \to \mathbb{C}\textsc{sp}$ recursively translates all outgoing transitions of a state. We map each state in the state machine (italic $s, t, \ldots$) to a CSP process (typewriter $\texttt{s}$,$\texttt{t}$,...) using unique names. We remind the reader that these are *variables* and thus placeholders for the identifier of a state or process.

Also, we need to consider that we need slightly different translations depending on the target mode of our translation: at least the state machine may be in the role of specification or implementation (sequence diagram vs. state machine, state machine vs. implementation, two implementations against each other). The specification is always translated using internal choice, while the implementation uses external choice for call-ins. Therefore, we use in the definitions below the choice operator $\prod$ to indicate that we choose angelic or demonic behaviour depending on the context we use the translation in. Thus we have:

$$\begin{aligned} &csp : \mathbb{C}tr_{SM} \to \mathbb{C}\textsc{sp} \\ &csp(Ctr) = \mathtt{P_{Init}} \cup csp_{sm}(Ctr.sm.i), \\ &\quad \mathtt{P_{Init}} = \prod_{\bar{c}\, \in Ctr.Init(Ctr.I.FDec_{priv})} \mathtt{sm_i}(\bar{c}) \qquad \text{(non-deterministic initialisation of } I.FDec_{priv}) \end{aligned}$$

where $\mathtt{P_{Init}}$ is a CSP process invoking the process corresponding to the initial state $\mathtt{sm_i}$ with given values $\bar{c}$ as per the specification if $I.init$ from the interface for the formal parameters $\overline{st} = I.FDec$, and

$$\begin{aligned}
&csp_{sm} : \mathbb{V} \to \mathbb{C}\textsc{sp} \\
&csp_{sm}(s) = \mathtt{s}(\overline{st}) \cup \bigcup_{t \in ts} csp_{sm}(t.t) && \text{(translation of transitive closure)} \\
&\quad \mathtt{s}(\overline{st}) = csp_{trans}(ts, \emptyset) && \text{(a process for each state)} \\
&\qquad ts = \{\langle s, m, t\rangle \mid \langle s, m, t\rangle \in sm.trans\} && \text{(outgoing transitions of } s\text{).}
\end{aligned}$$

In the translation of the outgoing transitions in csp_{trans}, we choose a translation that returns a single CSP process definition and handles both kinds of choice: as specification, we are allowed to call any method enabled by the protocol (internal choice), as implementation we have to accept every path the environment exercises (external choice). Guards have to be handled specially when translating with internal choice. The specification must only chose enabled courses of interaction, thus we turn each guard into an if-then-else over the guarding expression. A specification may still deadlock if no branch is enabled. The recursive construction below collects all enabled transitions in the specification and implementation and allows them to chose any enabled transition:

$$\begin{aligned}
&csp_{trans}(ts, ps) = \\
&\quad \begin{cases}
csp_{val}(m.g) \;\&\; csp(\langle s, m, t\rangle) & \text{iff } ts = \{\langle s, m, t\rangle\}, ps = \emptyset \\
(csp(\langle s, m, t\rangle) \prod ps) \lhd csp_{val}(m.g) \rhd (\prod ps) & \text{iff } ts = \{\langle s, m, t\rangle\}, ps \neq \emptyset \\
csp_{trans}(ts \backslash \{t\}, ps \cup \{csp(\langle s, m, t\rangle)\}) \lhd csp_{val}(m.g) \rhd csp_{trans}(ts \backslash \{t\}, ps), \\
\qquad\qquad t \in ts, \ \text{otherwise (push choice into branches).}
\end{cases}
\end{aligned}$$

The function csp_{val} for translating integer and boolean operations into CSP is omitted, since we do not handle navigation paths and method calls as we assume 'flat' rCOS as input. We only allow simple boolean and integer operations (and finite sets thereof). We do not consider the degenerate case of a contract without any methods. For $csp(\langle s, m, t\rangle)$, we handle the translation of a transition to a regular state in CSP:

$$csp(\langle s, m, t\rangle) = \mathtt{call\_}m?\bar{x} \to csp_{m}\left(m, \mathtt{ret\_}m!\bar{c} \to \mathtt{t}(\overline{st})\right)$$

We split the call and return of a method m into two different events $\mathtt{call\_}m$ and $\mathtt{ret\_}m$, so that the environment can synchronize on them. The return values $\bar{c}$ in the output event must be valid expressions over variables in the formal parameters $\overline{st}$ plus any locally declared variables. The process $\mathtt{t}$ is created by the previous definition of csp_{sm} since we translate all states in the transitive closure. Again choice resolves the non-determinism if there is more than one successor state (consider e.g. the predicate $\textsc{exmode}{=}true \sqcap \textsc{exmode}{=}false$ which allows progress with either of the values). Likewise, $csp_{val}(m.g)$ is the abstraction of an rCOS guard into CSP. In case of a deterministic *post*-mapping through $csp_{m}()$ below, it degenerates into a single branch.

The declaration of the CSP channels for events $\mathtt{call\_}m$ and $\mathtt{ret\_}m$ requires the mapping of 'flat' rCOS types of input and output/return parameters to CSP data types (currently only integers and boolean, for objects and set-like data structures see future work).

We define the translation of *a sequential composition of designs* mixed with CSP expressions, assuming right-associativity of the operator to simplify the rules: $d_1; d_2; d_3, \ldots = d_1; (d_2; (d_3; (\ldots) \ldots))$. We convert an imperative program with assignments into a functional style, where d_e is the continuation that has to be executed/translated at the end of a sequence:

$$
\begin{array}{lll}
\mathit{csp}_m(d, d_e) = & \text{match } d \text{ with} & \\
\mid \text{SKIP} & \longrightarrow \mathit{csp}_m(d_e, \text{SKIP}) \text{ if } d_e \neq \text{SKIP}, & \\
 & \quad\ \text{SKIP} \qquad\qquad\quad \text{else (end of translation branch)} & \\
\mid d_1; d_2 & \longrightarrow \mathit{csp}_m(d_1, d_2; d_e) & \\
\mid \text{Var } T\ x; d_v; \text{End } x & \longrightarrow \mathit{csp}_m(d_v, \text{SKIP}); \mathit{csp}_m(d_e, \text{SKIP}) & \\
 & \quad\ \text{(assignments introduce new scope)} & \\
\mid x := e & \longrightarrow \displaystyle\prod_{x \in \mathit{csp}_{val}(e)} \mathit{csp}_m(d_e, \text{SKIP}) & \text{(new scope for } x\text{)} \\
\mid \star(e, d_1) & \longrightarrow \mathtt{W}(\overline{st}), \text{where } \mathtt{W} \text{ is a fresh process name,} & \\
 & \quad\ \mathtt{W}(\overline{st}) = \mathit{csp}_m(d_1, \mathtt{W}(\overline{st})) \lhd \mathit{csp}_{val}(e) \rhd \mathit{csp}_m(d_e, \text{SKIP}) & \\
\mid [p \vdash R] & \longrightarrow \mathit{csp}_m(R, d_e) \lhd \mathit{csp}_{val}(p) \rhd \mathtt{STOP} & \\
\mid d_1 \lhd e \rhd d_2 & \longrightarrow \mathit{csp}_m(d_1, d_e) \lhd \mathit{csp}_{val}(e) \rhd \mathit{csp}_m(d_2, d_e) & \\
\mid d_1 \sqcap d_2 & \longrightarrow \mathit{csp}_m(d_1, d_e) \prod \mathit{csp}_m(d_2, d_e) & \\
\mid \text{CHAOS} & \longrightarrow \mathtt{CHAOS} & \\
\mid \mathtt{csp} & \longrightarrow \mathtt{csp}; \mathit{csp}_m(d_e, \text{SKIP}) & \text{(any CSP process)}
\end{array}
$$

The translation of contracts in $\mathbb{C}\mathit{tr}_{SeqD}$, contracts whose protocol is defined as a sequence diagram, is done by the function $\mathit{csp} : \mathbb{C}\mathit{tr}_{SeqD} \rightarrow \mathbb{C}\text{SP}$, which proceeds in a similar manner and exercises the same syntactical features of rCOS.

Verification of consistency and implementation. As we have shown in Fig. 1, we can now verify by using the FDR2 model checker that given a state machine sm and contract $\mathit{Ctr}_{SM} = (I, \mathit{Init}, \mathit{MSpec}, sm)$ and given a sequence diagram and a contract $\mathit{Ctr}_{SeqD} = (I, \mathit{Init}, \mathit{MSpec}, \mathit{seqd})$, we have $\mathit{csp}(\mathit{Ctr}_{SeqD}) \underset{M}{\parallel} \mathit{csp}(\mathit{Ctr}_{SM})$ is deadlock free, in which M is the set of generated CSP events (the method calls), that is, the traces specified in the sequence diagram are accepted by the state machine.

Before we can consider verifying an implementation against a contract, we discuss an abstraction framework that can be used to get a suitable over-approximation of an rCOS class into 'flat' rCOS that can be translated into CSP.

Integration. In [2], we describe how we integrate the state machine with a class containing the functionality specification to obtain a guarded design that we summarize in the following. We introduce a new variable *state* that holds a symbolic representation of the state that execution is currently in. Then, for each operation m we add a guard that only accepts the call-in into the method if we are in a state that has an outgoing transition labeled with m. Any additional guards on the transition (in the example the test ITEMCOUNTER < MAX) or on the design of m need to be respected as well. In the body, we update the state variable to the successor state.

We do not repeat the formalisation in [2] here, but rather use the example to illustrate the effect on the enterItem method. By integrating the class with the contract as given through the state machine we obtain:

```
public enterItem(Barcode code, int qty ; ) {
  ((state = 13928840) ∨ (state = 9528594) ∨ (state = 14696984) ∨
  ((itemCounter < max) ∧ (state = 4957896))) &
  VAR LineItem item ;
  [ pre  : store.catalog.find(c) != null ,
    post : line' = LineItem.new(c, q) ;
        line.subtotal' = store.catalog.find(c).price ∗ q;
             sale.lines.add(line)]  ;
  [ ⊢ itemCounter' = itemCounter + 1 ] ;
  if (state = 13928840) then {[ ⊢ state' = 9528594]}
  else { if (state = 9528594) then {[ ⊢ state' = 9528594]}
         else { if (state = 14696984) then {[ ⊢ state' = 4957896]}
                else { if (state = 4957896) then {[ ⊢ state' = 4957896]}
                       else { skip /∗ not reached ∗/}}}}}
```

The state identifiers have been automatically generated. Observe how enterItem has been used on four transitions, where one of them held the additional guard. Formally, we denote this integration of a contract and a class by *integrate*(*Ctr*, *C*) (compare with Fig. 1 again).

The Abstraction Process

In order to translate an rCOS specification to a CSP process, it may be necessary to abstract some parts of the designs. Indeed, a design usually contains functional specifications, which may not be relevant to the conformance to the protocol.

The design for enterItem(Barcode code, *int* qty) above contains information concerning both the protocol and the functional specification. Indeed, the variable item and the first pre/post-condition (checking if the code is in the store catalog and adding the item to the sale) have no concern with the protocol. It is then possible to 'remove' the statements related to item, or, in other words, to keep only those related to the variables concerned with the protocol, which are itemCounter, state and max for enterItem. Moreover, in the perspective of a 'flat' rCOS, all references to object with navigation paths (*i.e.* $x.y$ where $x \neq$ this) should also be removed. We first introduce the function $\mu : Exp \times 2^{\mathrm{VAR}} \rightarrow \mathbb{B}$, which indicates if an expression should be kept or not:

$$\begin{array}{lll}
\mu(e,l) = & \text{match } e \text{ with} & \\
\mid const & \rightarrow true & \\
\mid x & \rightarrow true & \text{if } x \in l \\
\mid x & \rightarrow false & \text{if } x \notin l \\
\mid path.x & \rightarrow true & \text{if } x \in l \wedge path = \text{this} \\
\mid path.x & \rightarrow false & \text{if } x \notin l \vee path \neq \text{this} \\
\mid \neg e_1 & \rightarrow \mu(e_1,l) & \\
\mid e_1 \text{ op } e_2 & \rightarrow \mu(e_1,l) \wedge \mu(e_2,l) & (\text{where op } \in \{\wedge,\vee,=,\neq,+,-,/,*\}) \\
\mid m(in;out) & \rightarrow false &
\end{array}$$

The abstraction process is an over-approximation, so every entity should refine its abstraction. We introduce the function $\alpha_p : Pred \times 2^{\mathrm{VAR}} \rightarrow Pred$ which abstracts every non atomic expression to *true*.

The designs are abstracted by the function $\alpha_k : D \times 2^{\mathrm{VAR}} \rightarrow D$:

$$\begin{array}{lll}
\alpha_k(d,l) = & \text{match } d \text{ with} & \\
\mid [p \vdash R] & \rightarrow [\alpha_p(p,l) \vdash \alpha_p(R,l)] & \\
\mid d_1 \lhd e \rhd d_2 & \rightarrow \alpha_k(d_1,l) \lhd e \rhd \alpha_k(d_2,l) & \text{if } \mu(e,l) \\
\mid d_1 \lhd e \rhd d_2 & \rightarrow \alpha_k(d_1,l) \sqcap \alpha_k(d_2,l) & \text{if } \neg\mu(e,l) \\
\mid d_1 \sqcap d_2 & \rightarrow \alpha_k(d_1,l) \sqcap \alpha_k(d_2,l) & \\
\mid d_1 ; d_2 & \rightarrow \alpha_k(d_1,l); \alpha_k(d_2,l) & \\
\mid \star(e,d_1) & \rightarrow \star(e, \alpha_k(d_1,l)) & \text{if } \mu(e,l) \\
\mid \star(e,d_1) & \rightarrow \text{Var bool } b; (b := \mathit{true} \sqcap b := \mathit{false}); & \\
 & \quad \star(b, \alpha_k(d_1,l); (b := \mathit{false} \sqcap b := \mathit{true})) & \text{if } \neg\mu(e,l) \\
\mid x := e & \rightarrow x := e & \text{if } x \in l \wedge \mu(e,l) \\
\mid x := e & \rightarrow \text{SKIP} & \text{if } x \notin l \vee \neg\mu(e,l) \\
\mid \mathit{path}.x := e & \rightarrow x := e & \text{if } x \in l \wedge \mu(e,l) \wedge \mathit{path} = \text{ this} \\
\mid \mathit{path}.x := e & \rightarrow \text{SKIP} & \text{if } x \notin l \vee \neg\mu(e,l) \vee \mathit{path} \neq \text{ this} \\
\mid \text{Var } T\ x & \rightarrow \text{Var } T\ x & \text{if } x \in l \\
\mid \text{Var } T\ x & \rightarrow \text{SKIP} & \text{if } x \notin l \\
\mid m(\mathit{in}, \mathit{out}) & \rightarrow \text{SKIP} & \\
\mid \text{SKIP}, \text{CHAOS} & \rightarrow \text{SKIP}, \text{CHAOS} &
\end{array}$$

This function removes every statement not only composed with variables given as a parameter and atomic expressions. Note that the conditional statement is asbtracted as a non deterministic choice if the condition is not atomic. In the same way, if the condition of a loop is not atomic, a new boolean variable b is introduced, and the design of the loop is extended to let the choice to set b to *true* or to *false*.

We extend the abstraction α_k to methods with the function $\alpha_m : \mathit{Meth} \times 2^{\mathrm{VAR}} \rightarrow \mathit{Meth}$ which, given a method and a list of variables, returns the method with the corresponding abstracted design.

Since the above design does not contain any reference to CODE or QTY, we can create a new method by removing these parameters from ENTERITEM. This abstraction is done by the function $\alpha_{par} : \mathit{Meth} \times 2^{\mathrm{VAR}} \rightarrow \mathit{Meth}$, defined by:

$$\alpha_{par}((\mathit{Name}, \mathit{in}, \mathit{out}, g, d), l) = (\mathit{Name}, \mathit{in}', \mathit{out}', g, d)$$

with $\mathit{in}' = \mathit{in} \setminus (l \setminus FV(D))$, $\mathit{out}' = \mathit{out} \setminus (l \setminus FV(D))$ and $FV(D)$ stands for the free variables in D. Note by removing only $(l \setminus FV(D))$, we ensure to remove only unused parameters. The functions α_k and α_{par} can be composed to define the function $\alpha : \mathit{Meth} \times 2^{\mathrm{VAR}} \rightarrow \mathit{Meth}$:

$$\alpha((\mathit{Name}, \mathit{in}, \mathit{out}, g, d), l) = \alpha_{par}(\alpha_m((\mathit{Name}, \mathit{in}, \mathit{out}, g, d), l), \mathit{in} \cup \mathit{out})$$

The method ENTERITEM can be abstracted by α, by keeping only the variables ITEMCOUNTER, STATE and MAX:

```
α(enterItem, {itemCounter, state, max}) = public enterItem() {
  ((state = 13928840) ∨ (state = 9528594) ∨ (state = 14696984) ∨
  ((itemCounter < max) ∧ (state = 4957896))) &
  [ ⊢ itemCounter' = itemCounter + 1 ] ;
  if (state = 13928840) then {[ ⊢ state ' = 9528594]}
  else { if (state = 9528594) then {[ ⊢ state ' = 9528594]}
else { if (state = 14696984) then {[ ⊢ state' = 4957896]}
       else { if (state = 4957896) then {[ ⊢ state' = 4957896]}
              else { skip /* not reached */}}}}}
```

Finally, we introduce the function $\alpha_C : C \times 2^{FDec} \to C$ which abstracts all the methods of a given class: $\alpha_C((FDec, MDec), l) = (l, \{\alpha(m, l) \mid m \in MDec\})$, (abstraction w.r.t. variables in l).

Now we are in a position to show that the integrated class still follows a contract: we first calculate the integrated specification from the contract $Ctr \in \mathbb{C}tr_{SM}$ and the implementation C, apply the appropriate abstraction function with respect to the interface to obtain a new flat rCOS specification *impl*. Then, we translate the abstracted methods into CSP and create a process that accepts a call to a method based on the current state through external choice, updates its state variables and starts over:

$$\begin{aligned} impl_{Ctr,C} &= \alpha_C(integrate(Ctr, C), Ctr.I.FDec_{priv}) \\ \mathtt{P}_C(state, \overline{st}) &= \mathop{\Box}_{m \in impl.MSpec} csp(m, \mathtt{P}_C((\mathtt{state}, \overline{st}))) \end{aligned}$$

We also need an initial process to start execution and have to state the initial state (derived from the state machine) and the initial values for the remaining attributes which are given in the contract, that is:

$$\mathtt{Init}_C = \prod \left\{ \left(\prod \{\mathtt{P}(\mathtt{i}, \overline{st}) \mid \overline{st} \in Ctr.Init(Ctr.I.FDec)\} \right) \mid i \in Ctr.sm.i \right\}$$

Then we check the trace refinement property of *impl* against $csp(Ctr)$. While the CSP generated from the contract (state machine) consists of a process for each state and a single method call can occur in various places, the overall structure of the CSP for the integrated specification is a single process allowing external choice over all possible guarded method calls, after which it will return to its main process again.

We could also apply the CSP translation of a whole class to an implementation we received from a third party to validate that it respects the contract, provided that we can find a suitable abstraction function.

4 Conclusion

We integrate the power of consistent UML modelling with the application of formal methods: a requirements model of a use case based on the rCOS methodology consists of hierarchical components, a protocol for an interface contract given as a state machine, the scenario to implement in the form of a sequence diagram, and the functionality specification in rCOS. Specification of behaviour in rCOS usually considers trace languages of method invocations instead of data.

The state machines and sequence diagrams can make use of the non-object oriented subset of rCOS designs, that we call 'flat' rCOS, for the functionality specification of operations. This seems like a very strong restriction in the age of object-oriented or even component-based development. But we show based on our example from the recent CoCoME [18] case study for a component-based point-of-sale system that this approach is already sufficient to ensure consistency of the specification between the sequence diagram and the state machine, and the state machine and an implementation (for example, it successfully detects an accidental inconsistency between an older version of the diagrams in [4]). We translate the artifacts into the process algebra CSP and use the model checker FDR2 to check the deadlock freedom. From our limited experience, as any counter example is a sequence of method invocations, it is easy to debug the model with regard to that particular trace. To get from a fully object-oriented rCOS specification of a class to 'flat' rCOS that is suitable for translation to CSP, the user employs a set of abstraction functions, like projection of a class onto a subset of its attributes.

As an illustration, we apply the technique from [2] to obtain an integrated specification in the form of a complete guarded design from the model and check it against the state machine.

We have implemented the rCOS modeler on top of the Eclipse Rich Client platform that can be used to graphically design a requirements model [5] and run the above transformation. Next, we intend to provide different abstractions for sets (e.g. precise, two- or three-valued) and for object instantiation.

In the larger scope, we plan semi-automated model transformation from a requirements model to a design- and component-model [21], proof support for refinement and abstractions, and code generation to provide an integrated software engineering solution for use case-driven models.

The Eclipse plugin of the rCOS modeler and the model used in this paper are freely available from `http://rcos.iist.unu.edu`.

Acknowledgements

This work was partially supported by the project HTTS funded by the Macao Science and Technology Development Fund, by the National Basic Research Program of China (973) under Grant No. 2005CB3218025 and NSFC under Grant No. 90612009.

References

1. Chen, X., He, J., Liu, Z., Zhan, N.: A model of component-based programming. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 191–206. Springer, Heidelberg (2007)
2. Chen, X., Liu, Z., Mencl, V.: Separation of concerns and consistent integration in requirements modelling. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, pp. 819–831. Springer, Heidelberg (2007)
3. Chen, Z., Hannousse, A.H., Hung, D.V., Knoll, I., Li, X., Liu, Y., Liu, Z., Nan, Q., Okika, J.C., Ravn, A.P., Stolz, V., Yang, L., Zhan, N.: Modelling with relational calculus of object and component systems–rCOS. In: Rausch et al [18], ch. 3

4. Chen, Z., Li, X., Liu, Z., Stolz, V., Yang, L.: Harnessing rCOS for tool support - the CoCoME Experience. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 83–114. Springer, Heidelberg (2007); UNU-IIST TR 383
5. Chen, Z., Liu, Z., Stolz, V.: The rCOS tool. In: Fitzgerald, J., Larsen, P.G., Sahara, S. (eds.) Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop. CS-TR-1099 in Technical Report Series, Newcastle University (May 2008)
6. Formal Systems (Europe) Ltd. FDR2 User Manual (2005)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
8. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall, Englewood Cliffs (1998)
9. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 3rd edn. Prentice-Hall Intl., Englewood Cliffs (2005)
10. Liu, Z., Mencl, V., Ravn, A.P., Yang, L.: Harnessing theories for tool support. In: Proc. of the Second Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006), pp. 371–382. IEEE Computer Society, Los Alamitos (2006); Full version as UNU-IIST Technical Report 343
11. Mellor, S., Balcer, M.: Executable UML: A foundation for model-driven architecture. Addison-Wesley, Reading (2002)
12. Möller, M., Olderog, E.-R., Rasch, H., Wehrheim, H.: Integrating a formal method into a software engineering process with UML and Java. Formal Aspects of Computing 20(2), 161–204 (2008)
13. Ng, M.Y., Butler, M.: Tool support for visualizing CSP in UML. In: George, C.W., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 287–298. Springer, Heidelberg (2002)
14. Ng, M.Y., Butler, M.: Towards formalizing UML state diagrams in CSP. In: SEFM, pp. 138–147. IEEE Computer Society, Los Alamitos (2003)
15. NoMagic, Inc. Magicdraw, http://www.magicdraw.com
16. Object Management Group. MDA Guide (2003), http://www.omg.org/cgi-bin/doc?omg/03-06-01
17. Pu, G., Stolz, V. (eds.): 1st Workshop on Harnessing Theories for Tool Support in Software. Electr. Notes in Theor. Comp. Sci., vol. 207. Elsevier, Amsterdam (2008)
18. Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.): The Common Component Modeling Example. LNCS, vol. 5153. Springer, Heidelberg (2008)
19. Roscoe, A.W.: Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1997)
20. Snook, C.F., Butler, M.J.: UML-B: Formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. 15(1), 92–122 (2006)
21. Yang, L., Stolz, V.: Integrating refinement into software development tools. In: Pu, Stolz [17], pp. 69–88
22. Yu, X., Wang, Z., Pu, G., Mao, D., Liu, J.: The verification of rCOS using Spin. In: Pu, Stolz [17], pp. 49–67

The Interplay between Relationships, Roles and Objects

Matteo Baldoni[1], Guido Boella[2], and Leendert van der Torre[3]

[1] Dipartimento di Informatica - Università di Torino - Italy
baldoni@di.unito.it
[2] Dipartimento di Informatica - Università di Torino - Italy
guido@di.unito.it
[3] University of Luxembourg
leendert@vandertorre.com

Abstract. In this paper we study the interconnection between relationships and roles. We start from the patterns used to introduce relationships in object oriented languages, and we show how the role model proposed in powerJava can be used to define roles. In particular, we focus on how to implement roles in an abstract way in objects representing relationships, and to specify the interconnections between the roles. Abstract roles cannot be instantiated. To participate in a relationship, objects have to extend the abstract roles of the relationship. Only when roles are implemented in the objects offering them, they can be instantiated, thus allowing another object to play those roles.

1 Introduction

The need of introducing the notion of relationship as a first class citizen in Object Oriented (OO) programming, in the same way as this notion is used in OO modelling, has been argued by several authors, at least since Rumbaugh [1]: he claims that relationships are complementary to, and as important as, objects themselves. Thus, they should not only be present in modelling languages, like ER or UML, but they also should be available in programming languages, either as primitives, or, at least, represented by means of suitable patterns.

Two main alternatives have been proposed by Noble [2] for modelling relationships by means of patterns:

- The relationship as attribute pattern: the relationship is modelled by means of an attribute of the objects which participate in the relationship. For example, the `Attend` relationship between a `Student` and a `Course` can be modelled by means an attribute `attended` of the `Student` and of an attribute `attendee` of the `Course`.
- The relationship object pattern: the relationship is modelled as a third object linked to the participants. A class `Attend` must be created and its instances related to each pair of objects in the relationship. This solution underlies languages introducing primitives for relationships, e.g., Bierman and Wren [3].

These two solutions have different pros and cons, as Noble [2] discusses. But they both fail to capture an important modelling and practical issue. If we consider the kind of

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 402–415, 2010.

examples used in the works about the modelling of relationships, we notice that relationships are also essentially associated with another concept: students are related to tutors or professors [3,4], basic courses and advanced courses [4], customers buy from sellers [5], employees are employed by employers, underwriters interact with reinsurers [2], *etc.* From the knowledge representation point of view, as noticed by ontologist like Guarino and Welty [6], these concepts are not natural kinds like person or organization. Rather, they all are *roles* involved in a relationship.

Roles have different properties than natural kinds, and, thus, are difficult to model with classes: roles can played by objects of different classes, they are dynamically acquired, they depend on other entities - the relationship they belong to and their players. Moreover, when an object of some natural type plays a certain role in a relationship, it acquires new properties and behaviors. For example, a student in a course has a tutor, he can take the exam and get a mark for the exam, another property which exists only as far as he is a student of that course.

We introduce roles in OO programming languages, in an extension of the Java programming language, called powerJava, described in [7,8,9,10]. The language powerJava introduces roles as a way to structure the interaction of an object with other objects calling their methods. Roles express the possibilities of interaction offered by the object to other ones (e.g., a `Course` offers the role `Student` to a `Person` which wants to interact with it), i.e., the methods they can call and the state of interaction. First, these possibilities change according to the class of the callers of the methods. Second, a role maintains the state of the interaction with a certain individual caller. As roles have a state and a behavior, they share some properties with classes. However, roles can be dynamically acquired and released by an object playing them. Moreover, they can be played by different types of classes. Roles in powerJava are essentially inner classes which are linked not only to an instance of the outer class, called institution, but also to an instance representing the player of the role. The player of the role, to invoke the methods of the roles it plays, it has to be casted to the role, by specifying both the role type and the institution it plays the role in (e.g., the course in which it is a student).

In [11] we add roles to the relationship as attribute pattern: the relationship is modelled as a pair of roles (e.g., attending a course is modelled by the role `Student` played by `Person` and `BasicCourse` played by `Course`) instead of a pair of links, like in the original pattern. In this way, the state of the relationships and the new behavior resulting from entering the relationship can be modelled by the fact that roles are adjunct instances with their state and behavior. However, that solution fails to capture the coordination between the two roles, since in this pattern the roles are defined independently in each of the objects offering them (`Person` offering `BasicCourse` and `Course` offering `Student`). This is essentially an encapsulation problem, raised by the presence of a relationship.

In this paper, we provide a solution to this limitation first by extending the relationship object pattern with roles, and then by introducing abstract roles defined by relationships and extended by roles of objects offering them. When roles are defined in the relationships, the interconnection between the roles can be specified (e.g., the methods describing the protocol the roles use to communicate). When roles are extended in the objects offering them, they can be customized to the context. Roles defined in the relationships are abstract and thus they cannot be instantiated. Roles can be instantiated only when they are extended in the objects which will participate to the relationship.

2 Roles and Relationships

Relations are deeply connected with roles. This is accepted in several areas: from modelling languages like UML and ER to knowledge representation discussed in ontologies and multiagent systems.

Pearce and Noble [12] notice that relationships have similarities with roles. Objects in relationships have different properties and behavior: "behavioural aspects have not been considered. That is, the possibility that objects may behave differently when participating in a relationship from when they are not. Consider again the student-course example [...]. In practice, a course will have many more attributes, such as a curriculum, than we have shown."

The link between roles and relationships is explicit in modelling languages like UML in the context of collaborations: a classifier role is a classifier like a class or interface, but "since the only requirement on conforming instances is that they must offer operations according to the classifier role, [...] they may be instances of any classifier meeting this requirement" [13]. In other words: a classifier role allows any object to fill its place in a collaboration no matter what class it is an instance of, if only this object conforms to what is required by the role. Classification by a classifier role is multiple since it does not depend on the (static) class of the instance classified, and dynamic (or transient) in the sense above: it takes place only when an instance assumes a role in a collaboration [14].

As noticed by Steimann [15], roles in UML are quite similar to the concept of interface, so that he proposes to unify the two concepts. Instead, there is more in roles than in interfaces. Steimann himself is aware of this fact: "another problem is that defining roles as interfaces does not cover everything one might expect from the role concept. For instance, in certain situations it might be desirable that an object has a separate state for each role it plays, even for different occurrences in the same role. A person has a different salary and office phone number per job, but implementing the Employee interface only entails the existence of one state upon which behaviour depends. In these cases, modelling roles as adjunct instances would seem more appropriate."

To do this, Steimann [16] proposes to model roles as classifiers related to relationships, but such that these classifiers are not allowed to have instances. In Java terminology, roles should be modelled as abstract classes, where some behavior is specified, but not all the behavior, since some methods are left to be implemented in the class extending them. These abstract classes representing roles should be then extended by other classes in order to be instantiated. However, given that in a language like Java multiple inheritance is not allowed, this solution is not viable, and roles can be identified with interfaces only.

In this paper, we overcome the problem of the lack of multiple inheritance, by allowing objects participating to the relationship to offer roles which inherit from abstract roles related to the relationship, rather than imposing that objects extend the roles themselves. This is made possible by powerJava.

```
class Printer {                     role User playedby UserReq
  private int printedTotal;            { void print();
  private void print(){...}              int getPrinted(); }

  definerole User {                 interface UserReq
                                       { String getName();
    private int printed;                 String getLogin();}

    public void print(){ ...        jack = new AuthPerson();
      printed = printed + pages;    laser1 = new Printer();
      Printer.print(that.getName()); laser1.new User(jack);
      }}}                           laser1.new SuperUser(jack);
                                    ((laser1.User)jack).print();
```

Fig. 1. A role `User` inside a `Printer`

3 Roles in powerJava

Baldoni *et al.* [10] introduce roles as affordances in powerJava, an extension of the object oriented programming language Java. Java is extended with:

1. A construct defining the role with its name, the requirements and the operations.
2. The implementation of a role, inside an object and according to its definition.
3. How an object can play a role and invoke the operations of the role.

Figure 1 shows the use of roles in powerJava. First of all, a role is specified as a sort of interface (`role` - right column) by indicating with an interface or class who can play the role (`playedby`) and which are the operations acquired by playing the role. Second (left column), a role is implemented inside an object as a sort of inner class which realizes the role specification (`definerole`). The inner class implements all the methods required by the role specification as it were an interface.

In the bottom part of the right column of Figure 1 the use of powerJava is depicted. First, the candidate player `jack` of the role is created. It implements the requirements of the roles (`AuthPerson` implements `UserReq` and `SuperUserReq`). Before the player can play the role, however, an instance of the object hosting the role must be created first (a `Printer laser1`). Once the `Printer` is created, the player `jack` can become a `User` too. Note that the `User` is created inside the `Printer laser1` (`laser1.newUser(jack)`) and that the player `jack` is an argument of the constructor of role `User` of type `UserReq`. Moreover `jack` plays the role of `SuperUser`.

The player `jack` to act as a `User` must be first classified as a `User` by means of a so-called *role casting* (`(laser1.User) jack`). Note that `jack` is not classified as a generic `User` but as a `User` of `Printer laser1`. Once `jack` is casted to its `User` role, it can exercise its powers, in this example, printing (`print()`). Such method is called a power since, in contrast with usual methods, it can access the state of other objects: namespace shares the one of the object defining the role. In the example, the method `print()` can access the private state of the `Printer` and invoke `Printer.print()`.

4 Relationship as Attribute with Roles Pattern

We first summarize how the relationship as attribute pattern is extended with roles in [11], and in the next section the relationship object pattern with roles. Then, starting from the limitation of these new patterns, in Section 6 we define a new solution introducing abstract roles in relationships. As an example we will use the situation where a `Person` can be a `Student` and follow a `Course` as a `BasicCourse` in his curriculum. Note that `BasicCourse` is not a subtype of `Course`, since a `Course` can be either a `BasicCourse` or an `BasicCourse` in different curricula.

In [11], the relationship as attribute pattern is extended with roles by reducing the relationship not only to two symmetric attributes `attended` and `attendees` but also to a pair of roles (see Figure 2). E.g., a `Person` plays the role of `Student` with respect to the `Course` and the `Course` plays the role of `BasicCourse` with respect to the `Person`.

The role `Student` is associated with players of type `Person` in the role specification (`role`), which specifies that a `Student` can take an exam (`takeExam`). Analogously, the role `BasicCourse` is associated with players of type `Course` in the role definition, which specifies that a `Course` can communicate with the attendee.

The role `Student` is implemented locally in the class `Course` and, viceversa, the role `BasicCourse` is defined locally in the class `Person`. Note that this is not contradictory, since roles describe the way an object offers interaction to another one: a `Student` represents how a `Course` allows a `Person` to interact with itself, and, thus, the role is defined inside the class `Course`. Moreover the behavior associated with the role `Student`, i.e., giving exams, modifies the state of the class including the role or calls its private methods, thus violating the standard encapsulation. Analogously, the `communicate` method of `BasicCourse`, modifies the state of the `Person` hosting the role by adding a message to the queue. These methods, in powerJava terminology, exploit the full potentiality of methods of roles, called *powers*, of violating the standard encapsulation of objects.

To associate a `Person` and a `Course` in the relationship, the role instances must be created starting from the objects offering the role, e.g. if
`Course c`: `c.new Student(p)`

When the player of a role invokes a method of a role, a power, it must be first role casted to the role. For example, to invoke the method `takeExam` of `Student`, the `Person` must first become a `Student`. To do that, however, also the object offering the role must be specified, since the `Person` can play the role `Student` in different instances of `Course`; in this case the `Course c`: `((c.Student)p).takeExam(...)`.

This pattern with roles allows state and behavior to be added to a relationship between `Person` and `Course`, without adding a new class representing the relationship. The limitation of this pattern is that the two roles `Student` and `BasicCourse` are defined independently in the two classes `Person` and `Course`. Thus, there is no warranty that they are compatible with each other (e.g., they communicate using the same protocol, despite the fact that they offer the methods specified in the role specification). Moreover, we would like that all roles of a relationship can access the private state of each other (i.e., share the same namespace). However, this would be feasible only if the

```
role Student playedby Person
   { int takeExam(String work); }
role BasicCourse playedby Course
   { void communicate(String text); }

class Person{
  String name;
  private Queue messages;
  private HashSet<BasicCourse> attended; //BasicCourses followed
  definerole BasicCourse {
    Person tutor;
    // method access the state of outer class
    void communicate (String text)
      { Person.messages.add(text); }
    BasicCourse(Person t){
      tutor=t;
      Person.attended.add(this); }}//add link

class Course {
  String code;
  String title;
  //students of the course
  private HashSet<Student> attendees;
  private int evaluate(String x){...}
  definerole Student {
    int number;
    int mark;
    int takeExam(String work)
     { return mark = Course.evaluate(work); }
    Student () //add link
      { Course.attendees.add(this); }}}
```

Fig. 2. Relationship as attribute with roles pattern in powerJava

two roles `Student` and `BasicCourse` are defined by the same programmer in the same context. This is not possible since the two player classes `Person` and `Course` may be developed independently.

The pattern has different pros and cons; the following list integrates Noble [2]'s discussions on them:

- It allows simple one-to-one relationships: it does not require a further class and its instance to represent the relationship between two objects.
- It allows a state and operations to be introduced into the objects entering the relationship, which was not possible without roles in the relationship as attribute pattern.
- It allows the integration of the role and the element offering it by means of powers.
- It allows us to show which roles can be offered by a class, and, thus, in which relationships they can participate, since they are all defined in the class.

```
class AttendBasicCourse{
  Student attendee;
  BasicCourse attended;
  static Hashset<AttendBasicCourse> all;

  definerole Student {
    int mark;
    int number;
    int takeExam(String work){
      mark= AttendBasicCourse.attended.evaluate(work);}
  }

  definerole BasicCourse {
    String program;
    Person tutor;
    private int evaluate(String work){...}
    void communicate(String t){
      //invoke the requirement of the player
      AttendBasicCourse.attendee.that.getMessage(t);}
    }

  AttendBasicCourse(Person p, Course c, String p, Person t){
    attendee = this.new Student(p);
    attended = this.new BasicCourse(c,p,t);
    AttendBasicCourse.all.add(this);
  }

  static void communicate(String text){
   for (AttendBasicCourse x: all) x.attended.communicate(text);}
}
```

Fig. 3. Relationship object with roles pattern, part I

Disadvantages of the relationship as attribute with roles pattern:

- It requires that the roles are already implemented offline inside the classes which participate in the relationship.
- It does not assure coherence of the pair of roles like student-course, buyer-seller, bidder-proponent, since they are defined separately in two different classes.
- The role cast to allow a player to invoke a power of its role requires to know the identity of the other participant in the relationship.
- It does not allow us to distinguish which is the role played in the other object participating in the relationship (e.g., a `Student` in the `attendees` set of a `Course` can follow the `Course` as a `BasicCourse` or an `AdvancedCourse`).

```
class Person{
 String name;
 Queue messages;
 void getMessage(String text) {messages.add(text)};
}
class Course {
  String code;
  String title;
}

role Student playedby Person
  { int takeExam(String work); }
role BasicCourse playedby Course
  { void communicate(String text); }

class University{
  public static void main (String[] args){
    Person p = new Person();
    Course c = new Course();
    a = new AttendBasicCourse(p,c,program,tutor);
    //p as a Student of Course takes the exam
    ((a.Student)p).takeExam(work);
    //c’s message to  Student of  Course
    ((a.BasicCourse)c).communicate(text);}}
```

Fig. 4. Relationship object with roles pattern, part II

5 Relationship Object Pattern

The alternative relationship object with roles pattern introduces an AttendBasicCourse class modelling the relationship between Person and Course. However, the AttendBasicCourse class is not linked to a Person and a Course. Rather, the Person plays the role Student in the class AttendBasicCourse and the Course the role BasicCourse (see Figures 3, 4 and the UML diagram in Figure 5)[1]. Like in the previous solution the roles are modelled as inner classes. In this example, the roles are implemented in the class AttendBasicCourse. Its instances contain the properties and behaviors added when instances of Person and Course, respectively, participate in the relationship. Additionally, properties and behaviors which are associated to the relationship itself, like entering in the relationship and constraints on the participants can be added to the relationship class.

To relate a Person and a Course in a relationship, an instance of AttendBasicCourse must be created, together with an instance of Student played by the Person and of BasicCourse played by the Course. To invoke a

[1] The arrow starting from a crossed circle, in UML, represents the fact that the source class can be accessed by the arrow target class.

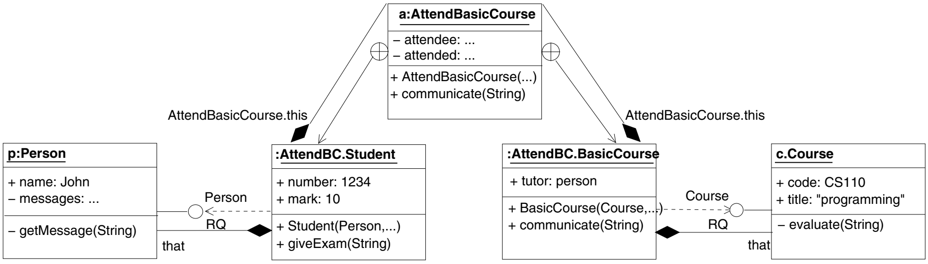


Fig. 5. The UML representation of the relationship object with roles pattern example

power of `Student`, a `Person` must be role casted to the role `Student` starting from an instance of the class `AttendBasicCourse`.
Advantages of the relationship role object with roles pattern:

- It allows a state and operations of the relationship to be introduced besides the state and operations added to the objects entering the relationship.
- It allows listing all instances of the relationship and centralize operations like entering the relationship and to check constraints on the relationship.
- It enforces to create both role instances at the same time, since they are linked to the same relation instance, thus avoiding the risk of inconsistencies.
- It allows the integration of the role with the relationship and with the other role, since the powers of a role can access both. In this way it is possible to deal with coordination issues [7].
- To make a role cast it is necessary only to know the relationship instance, thus, the other participant can change without notice.
- It does not require that the classes of players already implement the role classes. To play a role it is sufficient to satisfy the requirements.

Disadvantages of the relationship object with roles pattern:

- It requires a further class and its instance.
- It does not allow the integration of roles with the objects offering them (e.g., `Student` is defined separately of the class `Course`, which, as a consequence, cannot be accessed). Thus, to play a role, an object is required to offer additional methods (see `getMessage` in Figure 3).

6 Abstract Roles and Relationships

From the above discussion, the following requirements emerge:

- to define the interaction between the roles separately from the classes offering them to participate in the relationship. This guarantees that the interaction between the objects eventually playing the roles is performed in the desired way;
- that the roles of a relationship have access to the private state of each other to facilitate their programming;

- that the roles also have access to the private states of the objects offering them (like in powerJava) to customize them to the context.

These requirements mirror the complexities concerning encapsulation, which arise when relationships are taken seriously, as noticed by Noble and Grundy [5].

A solution to the encapsulation problem is possible in powerJava by exploiting an often disregarded feature of Java. Inner classes share the namespace of the outer classes containing them. When a class extends an inner class in Java, it maintains the property that the methods defined in the inner class which it is extending continue to have access to the private state of the outer class instance containing the inner class. If the inner class is extended by another inner class, the resulting inner class belongs to the namespaces of both outer classes. Moreover, an instance of such an inner class has a reference to both outer class instances so to be able to access their states. The possible ambiguities of identifiers accessible in the two outer classes and in the superclass are resolved by using the name of the outer class as a prefix of the identifier (e.g., `Course.registry`).

This feature of Java, albeit esoteric, has a precise semantics, as discussed by [17].

The new solution we propose allows the introduction of a new class representing the relationship as in the relationship object with roles pattern, and to define the roles inside it. The idea is illustrated in Figure 8 as an UML diagram.

First, as in the relationship object with roles pattern, a class for creating relationship objects is created (e.g., `AttendBasicCourse`): it will contain the implementation of the roles involved in the relationship (e.g., `Student` and `BasicCourse` in `AttendBasicCourse`), see Figure 6. The interaction between the roles is defined at this level since the powers of each role can access the state of the other roles and of the relationship.

These roles must be defined as abstract and so they cannot be instantiated. Moreover, the methods containing the details about the customization of the role can be left unfinished (i.e., declared as abstract) if they need to be completed depending on the classes offering the roles which extend the abstract roles.

Second, the same roles in the relationship can be implemented in the classes participating in the relationship (and, thus, they can be extended separately), accordingly to the relationship as attribute pattern, see Figure 7 (`Person` offering `BasicCourse` and `Course` offering `Student`). However, these roles (e.g., `Student` and `BasicCourse`), rather than being implemented from scratch, extend the abstract roles of the relationship object class (e.g., `AttendBasicCourse`), filling the gaps left by abstract methods in the abstract roles (both public and protected methods). The extension is necessary to customize the roles to their new context. Methods which are declared as final in the abstract roles cannot be overwritten, since they represent the interaction among roles in the scope of the relationship. Further methods can be declared, but they are not visible from outside since both the abstract role and the concrete one have the signature of the role declaration.

Note that the abstract roles are not extended by the classes participating in the relationship (e.g., `Course` and `Person`), but by roles offered by (i.e., implemented into) these classes (e.g., `Student` and `BasicCourse`). Otherwise, the classes participating in the relationship could not extend further classes, since Java does not allow multiple inheritance, thus limiting the code reuse possibilities.

```
role Student playedby Person
  { int takeExam(String work); }

role BasicCourse playedby Course
  { void communicate(String text); }

class AttendBasicCourse {
  Student attendee;
  BasicCourse attended;

  abstract definerole Student {
    int mark;
    int number;
    //method modelling interaction
    final int takeExam(String work){
      return mark = evaluate(work);}
    //method to be implemented which is not public
    abstract protected int evaluate(String work);
  }

  abstract definerole BasicCourse {
    String program;
    Person tutor;
    //method to be implemented which is public
    abstract void communicate(String text);
  }

  AttendBasicCourse(String pr, Person t){
    attendee = c.new Student(p,this);
    attended = p.new BasicCourse(c,this,t);
  }
}
```

Fig. 6. Abstract roles

The advantage of these solution is that roles can share both the namespace of the relationship object class and the one of the class offering the roles, as we required above. This is possible since extending a role implementation is the same as extending an inner class in Java: roles are compiled into inner classes by the powerJava precompiler.

Based on this idea we propose here a limited extension of powerJava, which allows abstract roles to be defined inside relationship object classes, and to let standard roles extend them. The resulting roles will belong both to the namespace of the class offering them and to the relationship object class. Moreover, the resulting roles will inherit the methods of the abstract roles.

Note that the abstract roles cannot be instantiated. This is so that the are used only to implement both the methods which define the interaction among the roles, and the methods which are requested to be contextualized. The former will be final methods which are inherited, but which cannot be overwritten in the eventual extending role: they will access the state and methods of the outer class and of the sibling roles. The latter will

```
class Course {
  String code, title;
  private HashSet<Student> attendees;

  definerole Student extends AttendBasicCourse.Student {
    Student() {
      Course.this.attendee = this;
    }
    //abstract method implementation
    protected int evaluate(String work)
      { /*Course specific
           implementation of the method */  }  } }

class Person {
  String name;
  private Queue messages;
  private HashSet<BasicCourse> attended;
  //courses followed as BasicCourse

  definerole BasicCourse extends AttendBasicCourse.BasicCourse {
    BasicCourse(Person t) {
      tutor=t;
      Person.this.attended=this; }
      //abstract method implementation
      void communicate (String text)
        {Person.this.messages.add(text);
        } } }
```

Fig. 7. Abstract roles extended

be abstract protected methods, which are used in the final ones, and which must be implemented in the extending class to tailor the interaction between the abstract role and the class offering the role. If these methods are declared as protected they are not visible outside the package. These methods have access to the class offering the extending roles.

Besides adding the property `abstract` to roles, three other additions are necessary in powerJava.

First, we add an additional constraint to powerJava: if a role implementation extends an abstract role, it must have the same name. Thus, the abstract and concrete role have the same requirements. Moreover, it is only possible to extend abstract roles, while general inheritance among roles is not discussed here.

Second, the methods of the abstract role can make reference to the outer class of the extending role. This is realized by means of a reserved variable `outer`, which is of type `Object` since it is not possible to know in advance which classes will offer the extended role. This variable is visible only inside abstract roles.

Third, to create a role instance it is also necessary to have at disposal the relationship object offering the abstract roles, and the two roles must be created at the same time.

For example, the constructor of a relationship:

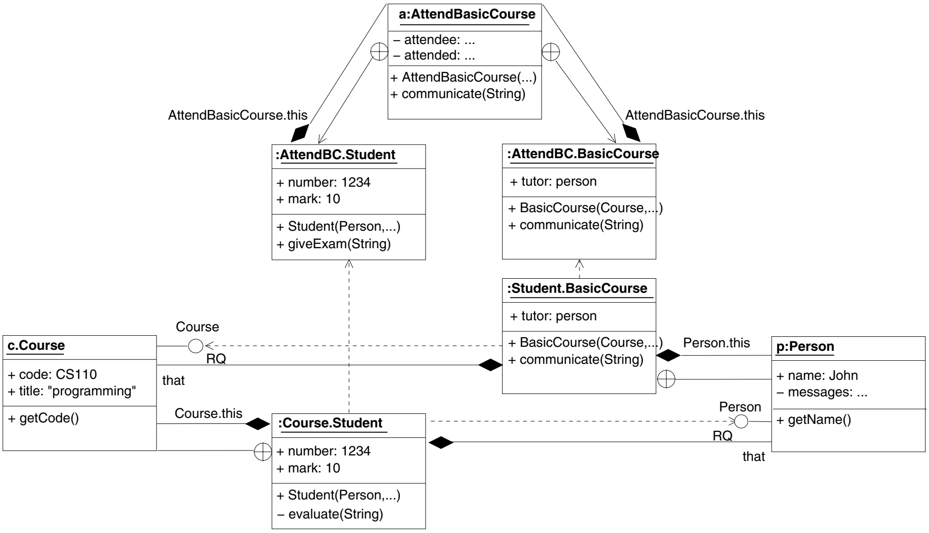


Fig. 8. The UML representation of the new relationship pattern

```
AttendBasicCourse(Person p, Course c){//...
  c.new Student(p,this);
  p.new BasicCourse(c,this); }
```

Where `Student` and `BasicCourse` are the class names of the concrete roles implemented in `p` and `c` and they are the same as the abstract roles defined in the relation.

The types of the arguments `Person` and `Course` are the requirements of the roles `Student` and `BasicCourse` which will be used to type the `that` parameter referring to the player of the role.

Moreover, the first and the second argument of the constructor are added by default: the first one represents the player of the role, while the second one, present only in roles extending abstract roles, is the reference to the relationship object. This is necessary since the inner class instance represented by the role has two links to the two outer class instances it belongs to. This reference is used to invoke the constructor of the abstract role, as required by Java inner classes. For example, the constructor of the role `Course.Student` is the following one.

```
Student(Person p, AttendBasicCourse a){ a.super(); //... }
```

However, these complexities are hidden by powerJava which adds the necessary parameters and code during precompilation.

The entities related by the relationship must preexist to it:

```
Person p = new Person();
Course c = new Course();
AttendBasicCource r = new AttendBasicCourse(p,c);
((c.Student)p).takeExam(w);
((p.BasicCourse)c).communicate(text);
```

7 Conclusion

In this paper we discuss how abstract roles can be introduced when relationships are modelled in OO programs: first abstract roles are defined in the relationship object class, which specify the interaction, and then the abstract roles are extended in the classes offering them. This pattern solves the encapsulation problems raised when relationships are introduced in OO.

We introduce abstract roles using the language powerJava, a role endowed version of Java ([7,8,9,10,11]).

References

1. Rumbaugh, J.: Relations as semantic constructs in an object-oriented language. In: Procs. of OOPSLA, pp. 466–481 (1987)
2. Noble, J.: Basic relationship patterns. Pattern Languages of Program Design, vol. 4. Addison-Wesley, Reading (2000)
3. Bierman, G., Wren, A.: First-class relationships in an object-oriented language. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 262–286. Springer, Heidelberg (2005)
4. Albano, A., Bergamini, R., Ghelli, G., Orsini, R.: An object data model with roles. In: Procs. of Very Large DataBases (VLDB 1993), pp. 39–51 (1993)
5. Noble, J., Grundy, J.: Explicit relationships in object-oriented development. In: Procs. of TOOLS 18 (1995)
6. Guarino, N., Welty, C.: Evaluating ontological decisions with ontoclean. Communications of ACM 45(2), 61–65 (2002)
7. Baldoni, M., Boella, G., van der Torre, L.: Roles as a coordination construct: Introducing powerJava. Electronic Notes in Theoretical Computer Science 150, 9–29 (2006)
8. Baldoni, M., Boella, G., van der Torre, L.W.N.: Modelling the interaction between objects: Roles as affordances. In: Lang, J., Lin, F., Wang, J. (eds.) KSEM 2006. LNCS (LNAI), vol. 4092, pp. 42–54. Springer, Heidelberg (2006)
9. Baldoni, M., Boella, G., van der Torre, L.: Interaction among objects via roles: sessions and affordances in powerjava. In: Procs. of PPPJ 2006, pp. 188–193. ACM, New York (2006)
10. Baldoni, M., Boella, G., van der Torre, L.: Interaction between Objects in powerJava. Journal of Object Technology 6, 7–12 (2007)
11. Baldoni, M., Boella, G., van der Torre, L.: Relationships meet their roles in object oriented programming. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 440–448. Springer, Heidelberg (2007)
12. Pearce, D., Noble, J.: Relationship aspects. In: Procs. of AOSD, pp. 75–86 (2006)
13. OMG: OMG Unified Modeling Language Specification, Version 1.3 (1999)
14. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley, Reading (1999)
15. Steimann, F.: A radical revision of UML's role concept. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 194–209. Springer, Heidelberg (2000)
16. Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. Data and Knowledge Engineering 35, 83–848 (2000)
17. Smith, M., Drossopoulou, S.: Inner classes visit aliasing. In: ECOOP 2003 Workshop on Formal Techniques for Java-like Programming (2003)

A Coordination Model for Interactive Components

Marco A. Barbosa, Luis S. Barbosa, and José C. Campos

Department of Informatics & Computer Science and Technology Center (CCTC)
Minho University, Portugal
{marco,lsb,jfc}@di.uminho.pt

Abstract. Although presented with a variety of 'flavours', the notion of an *interactor*, as an abstract characterisation of an interactive component, is well-known in the area of formal modelling techniques for interactive systems. This paper replaces traditional, hierarchical, 'tree-like' composition of interactors in the specification of complex interactive systems, by their *exogenous coordination* through general-purpose software *connectors* which assure the flow of data and the meet of synchronisation constraints. The paper's technical contribution is twofold. First a modal logic is defined to express behavioural properties of both interactors and connectors. The logic is new in the sense that its modalities are indexed by fragments of *sets* of actions to cater for action co-occurrence. Then, this logic is used in the specification of both interactors and coordination layers which orchestrate their interconnection.

Keywords: Interactors, coordination models.

1 Introduction

Modern interactive systems resort to increasingly complex architectures of user interface components. With the generalisation of ubiquitous computing, the notion of interactive system itself changed. Single interactive devices have been replaced by frameworks where devices are combined to provide services to a number of different, often anonymous, users accessing them in a competing way. This may explain the increasing interest on rigorous methodologies to develop useful, workable models of such systems. In such a setting, the concept of an *interactor* was originally proposed by Faconti and Paternò [13], as an abstraction for a graphical object capable of both input and output, typically specified in a process algebra. This was further generalised by Duke and Harrison [12] for modelling interactive systems. Interactors become able not only to communicate through i/o ports, but also to convey information about their state through a rendering relation that maps the latter to some presentation medium.

The framework outlined in [12], however, does not prescribe a specification notation for the description of interactor state and behaviour. Several possibilities have been considered. One of them, which directly inspired this piece of research, was developed by the third author in [8] and resorts to Modal Action

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 416–430, 2010.

Logic (MAL) [19] to specify behavioural constraints. Another one [18] uses LOTOS to express a relation between input and output ports. Actually, the notion of an interactor as a structuring mechanism for formal models of interactive systems, has been an influential one. It has been used, for example, with LOTOS [13,17], Lustre [10], Petri nets [6], Higher Order Processes [11], or Modal Action Logic [9].

Whatever the approach, modelling complex interactive systems entails creating architectures of interconnected interactors. In [8] such models are built hierarchically through 'tree-like' aggregation. Composition is typically achieved by the introduction of additional axioms and/or dedicated interactors to express the control logic for communication. This, in turn, adds dramatically to the complexity of the proposed models. Moreover, it does not promote a clear separation of concerns between modelling interactors and the specification of how they interact with each other.

This is exactly the point where the contribution of this paper may be placed. We adopt an *exogenous coordination* approach to the composition of interactors which entails an effective separation of concerns between the latter and the specification of how they are organised into specific architectures and interact to achieve common goals. Exogenous coordination draws a clear distinction between the *loci* of computational effort and that of interaction control, the latter being blind with respect to the structure of values and procedures which typically depend on the application domains.

Our approach is based on previous work on formal calculi for component coordination published in [4,2] and closely inspired by Arbab's Reo model [1]. In this paper we propose a particular model wherein complex coordinators, called *connectors*, are compositionally built out of simpler ones. This implies that not only should it be generally possible to produce different systems by composing the same set of interactors in different ways, but also that the difference between two systems composed out of the same set of interactors must arise out of their composition shemes, *i.e.*, their glue code.

Research reported here is a follow-up of a previous attempt to use the coordination paradigm to express the logic governing the composition of interactors, reported in [3], where a process algebra framework was used to specify connector's behavioural constraints. This, however, proved difficult to smoothly combine with interactors whose evolution is typically given by modal assertions. In this paper an extension to Hennessy-Milner logic [14] is proposed to express behavioural properties of both interactors and connectors. The novelty in the logic is the fact that its modalities are indexed by *sets* of actions to cater for action co-occurrence. Moreover, modalities are interpreted as asserting the existence of transitions which are indexed by a set of actions of which only a subset may be known. Both co-occurrence and such a sort of partial information about transitions seem to be essential for software coordination.

The rest of the paper is organised as follows. Section 2 introduces modal language $\mathbb{M}$, which is used to specify interactors in section 3, and software connectors in section 4. Section 5 brings interactors and the coordination layer together through the notion of a *configuration*. A few examples are discussed to assess the merits of proposed approach. Finally, a few topics for future work are discussed in section 6.

2 A Logic for Behaviour

2.1 A Modal Language

Like many other computing artefacts, both interactors and connectors exhibit *reactive* behaviour. They evolve through reaction, either to internal events (*e.g.*, an alarm timeout) or to the accomplishment of interactions with environment (*e.g.*, the exchange of a datum in a channel end). Following a well established convention in formal modelling, we refer to all such reaction points simply as *actions*, collected on a denumerable set Act. Then we define modal operators which qualify the validity of logical formaluæ with respect to action occurrence, or, more generally, to action *co-occurrence*.

Having mechanisms to express *co-occurrence* becomes crucial in modelling coordination code. For example, what characterises a *synchronous channel*, the most elementary form of software glue to connect two running interactors, is precisely the fact that any interaction in its input end is simultaneous with another interaction in the output end. Note that temporal simultaneity is understood here as *atomicity*: simultaneous actions cannot be interrupted.

The modal language introduced in the sequel is similar to the well-known Hennessy-Milner logic [14], but for a detail which makes it possible to express (and reason about) action *co-occurrence*. The basic idea is that a formula like $\langle a \rangle \phi$, for $a \in Act$, which in [14] asserts the existence of a transition indexed by a leading to a state which verifies assertion ϕ, is re-interpreted by replacing 'indexed by a' by 'indexed by a set of actions of which a is part of'. Therefore, modalities are relative to sets of actions, whose elements are represented by juxtaposition, regarded as *factors* of a (eventually larger) compound action.

In detail, modalities are indexed by either *positive* or *negative* action *factors*, denoted by K and $\sim\!K$, for $K \subseteq Act$, respectively. Intuitively, a *positive* (respectively, *negative*) factor refers to transitions whose labels include (respectively, exclude) all actions in it. Annotation $\sim$ may be regarded as an involution over $\mathcal{P}(Act)$ (therefore, $\sim\sim\!K = K$).

Formally $\mathbb{M}$ has the following syntax, where W is a positive or negative action factor and Ψ ranges over elementary propositions,

$$\phi \ ::= \Psi \mid \mathsf{true} \mid \mathsf{false} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid \langle W \rangle \phi \mid [W]\phi$$

Its semantics is given by a satisfiability relation wrt to system's states. For the non modal part this is as one would expect: for example $s \models \mathsf{true}$, $s \not\models \mathsf{false}$ and $s \models \phi_1 \wedge \phi_2 \Leftrightarrow s \models \phi_1 \ \wedge \ s \models \phi_2$. For the modal connectives, we define

$$s \models \langle W \rangle \phi \Leftrightarrow \langle \exists\ s' \ : \ \langle \exists\ \theta \ : \ s \xrightarrow{\theta} s' : \ W \prec \theta \rangle : \ s' \models \phi \rangle$$

$$s \models [W]\phi \Leftrightarrow \langle \forall\ s' \ : \ \langle \exists\ \theta \ : \ s \xrightarrow{\theta} s' : \ W \prec \theta \rangle : \ s' \models \phi \rangle$$

where

$$W \prec X \;\triangleq\; \begin{cases} W = K, \text{for } K \subseteq \mathit{Act} \;\;\Rightarrow\;\; K \subseteq X \\ W = \sim K, \text{for } K \subseteq \mathit{Act} \;\;\Rightarrow\;\; K \not\subseteq X \end{cases}$$

For example, if there exists a state s' such that $s \xrightarrow{abcd} s'$ and s' verifies some formula ϕ, then $s \models \langle bd \rangle \phi$. Dually, assertion $[\sim abc]\mathsf{false}$ states that all transitions whose labels do not involve, at least and simultaneously, actions in set $\{a, b, c\}$ lead to states which validate false and their occurrence is, therefore, impossible.

Modal connectives can be extended to *families* of both 'positive' or 'negative' action factors as follows:

$$s \models \langle F \rangle \phi \Leftrightarrow \langle \exists\, W \;:\; W \in F :\; \langle W \rangle \phi \rangle$$
$$s \models [F] \phi \Leftrightarrow \langle \forall\, W \;:\; W \in F :\; [W] \phi \rangle$$

where $F \subseteq (\mathcal{P}(\mathit{Act}) \cup \sim\mathcal{P}(\mathit{Act}))$. Just as actions in an action factor are represented by juxtaposition, as in $\langle abc \rangle$, action factors in a family thereof are separated by commas, as in $\langle J, K, L \rangle$. Set complement to $\mathcal{P}(\mathit{Act}) \cup \sim\mathcal{P}(\mathit{Act})$ is denoted by symbol $-$ as in $[-K]\mathsf{false}$ or $\langle - \rangle \mathsf{true}$, the latter abbreviating $-\emptyset$. The first assertion states that only transitions exactly labelled by factor K can occur. The second one that there exists, from the current state, at least a possible transition (of which no particular assumption is made).

Most results on Hennessy-Milner logic carry over $\mathbb{M}$. In particular, it can be shown that modal equivalence in $\mathbb{M}$ entails bisimulation equivalence for processes in CCS-like calculus extended with action co-occurrence. Although this is not the place to explore the structure of $\mathbb{M}$, the following *extension* laws are needed in the sequel: for all $a, a' \in \mathit{Act}$, $K, K' \subseteq \mathit{Act}$,

$$[a]\phi \;\Leftarrow\; [aa']\phi \quad \text{and} \quad \langle a \rangle \phi \;\Leftarrow\; \langle aa' \rangle \phi \tag{1}$$
$$[K]\phi \;\Leftarrow\; [K, K']\phi \quad \text{and} \quad \langle K \rangle \phi \;\Rightarrow\; \langle K, K' \rangle \phi \tag{2}$$
$$[K]\phi \wedge [K']\phi \;\Leftrightarrow\; [K, K']\phi \tag{3}$$
$$\langle K \rangle \phi \vee \langle K' \rangle \phi \;\Leftrightarrow\; \langle K, K' \rangle \phi \tag{4}$$

Proofs proceed by unfolding definitions. For example, the first part of (1) is proved as follows:

$$\begin{array}{ll} & s \models [a]\phi \\ \Leftrightarrow & \quad \{ \text{ definition } \} \\ & \langle \forall\, s' \;:\; \langle \exists\, \theta \;:\; s \xrightarrow{\theta} s' :\; \{a\} \subseteq \theta \rangle :\; s' \models \phi \rangle \\ \Leftarrow & \quad \{ \text{ set inclusion } \} \\ & \langle \forall\, s' \;:\; \langle \exists\, \theta \;:\; s \xrightarrow{\theta} s' :\; \{a, a'\} \subseteq \theta \rangle :\; s' \models \phi \rangle \\ \Leftrightarrow & \quad \{ \text{ definition } \} \\ & s \models [aa']\phi \end{array}$$

It is also easy to see that, for K and K', both positive or both negative,

$$[K, K']\phi \Rightarrow [K \cup K']\phi \tag{5}$$
$$\langle K, K'\rangle\phi \Leftarrow \langle K \cup K'\rangle\phi \tag{6}$$

2.2 Typical Properties

To exemplify the use of the logic and introduce some notation to be used in the sequel, let us consider a number of properties useful for the specification of both interactors and coordination schemes. Most of the latter are designed to preclude interactions in which some action factor K is absent. This leads to the following property schemes

$$\mathsf{only}\, K \;\triangleq\; [\sim\! K]\mathsf{false} \quad \text{and} \quad \mathsf{forbid}\, K \;\triangleq\; \mathsf{only}\; \sim\! K$$

Properties above entails conciseness in expression. For example, assertion $\mathsf{only}\, K \wedge \mathsf{only}\, L \wedge \mathsf{forbid}\, M$ abbreviates, by (3), to $\mathsf{only}\, K, L, \sim\! M$. A dual property asserts the existence of at least a transition of which a particular action pattern is a factor, *i.e.*, $\mathsf{perm}\, K \;\triangleq\; \langle K\rangle\mathsf{true}$. Or, not only possible, but also mandatory, $\mathsf{mandatory}\, K \;\triangleq\; \langle -\rangle\mathsf{true} \wedge \mathsf{only}\, K$.

More complex patterns of behaviour are expressed by nesting modalities, as in $[K]\langle L\rangle\phi$, which expresses a sort of invariant: after every occurrence of an action with factor K, there is, at least, a transition labelled by actions in L which validates ϕ. The complement of $\langle -\rangle\mathsf{true}$ is $[-]\mathsf{false}$ which asserts no transition is possible. Notice that their duals — $\langle -\rangle\mathsf{false}$ and $[-]\mathsf{true}$ — are just abbreviations of constants false and true, respectively.

3 $\mathbb{M}$-Interactors

3.1 A Language for $\mathbb{M}$-Interactors

As stated in the Introduction, our aim is to use a single specification notation for both *interactors*, which, in this setting, correspond to the computational entities, and *connectors*, which cater for the coordination of the former. Modal language $\mathbb{M}$ is, of course, our candidate for this double job — this section focuses on its first part.

The definition of a $\mathbb{M}$-interactor is adapted from [12], but for the choice of the behaviour specification language. Formally,

Definition 1. *An interactor signature is a triple* (S, α, Γ)*, where* S *is a set of sorts,* α *a* S*-indexed family of attribute symbols, and* Γ *a set of action symbols. An* $\mathbb{M}$*-interactor is a tuple* $(\Delta, \rho, \gamma, Ax_\Delta)$ *where* Δ *is an interactor signature,* $\rho : \mathbb{P} \longleftarrow \alpha$ *and* $\gamma : \mathbb{P} \longleftarrow \Gamma$ *are rendering relations, from attributes and actions, respectively, to some presentation medium* $\mathbb{P}$*, and* Ax_Δ *a set of axioms over* Δ *expressed in the* $\mathbb{M}$ *language.*

The set of ports provided by an interactor is defined by ρ, γ, and Γ. Ports induced by ρ are output ports used to read the value of attributes and are always available. This condition is expressed by $\langle\forall\ p\ :\ p \in \mathsf{range}\,\rho :\ \langle p\rangle\mathsf{true}\rangle$. Ports in Γ are input/output ports and their availability is governed by axioms in Ax_Δ.

Syntactically, the definition of an interactor has three main declarations: of attributes, actions and axioms. The first two define the signature. The rendering relation is given by annotations on the attributes. Actions can also be annotated to assert whether or not that they are available to the user. Fig. 1 shows a very simple example of an interactor modelling an application window. Two attributes are declared, indicating whether the window is visible or displays new information.

interactor *window*
  attributes
    $\boxed{\text{vis}}$ *visible*, *newinfo* : *bool*
  actions
    hide show update invalidate
  axioms
    $[hide]\ \neg visible$
    $[show]\ visible$
    $[update]\ newinfo$
    $[invalidate]\ \neg newinfo$
    forbid *hide show*
    forbid *update invalidate*

Fig. 1. A window interactor

Available actions model the change of visibility and information displayed in the window. Their effect in the state of the interactor is defined by the axioms in the figure. In this example, the rendering relation is defined by the $\boxed{\text{vis}}$ annotation, which indicates that all attributes are (visually) perceivable.

Although the behavioural properties specified in this example are rather simple, in general, it is necessary to specify when actions are permitted or required to happen. This is achieved with the perm and mandatory assertions, typically stated in a guarded context. Thus,

- perm $K \rightarrow \Phi$, where Φ is a non modal proposition over the state space of the interactor, as perceived by the values of its attributes. The assertion means that *if actions containing action factor K are permitted then Φ evaluates to* true.
- $\Phi \rightarrow$ mandatory K, meaning *actions containing action factor K are inevitable whenever Φ evaluates to* true.

A useful convention establishes that permissions, but not obligations, are asserted by default. I.e., by default anything can happen, but nothing must happen. This facilitates makes adding or removing permissions and obligations incrementally when writing specifications.

3.2 Composing Interactors

In the literature, and specifically in [8], interactors are composed in the 'classical' way, *i.e.*, by a specification *import* mechanism, illustrated below by means of a small example. In the literature, and specifically in [8], interactors are

interactor *space*
  attributes
    $\boxed{\text{vis}}$ $state : \{open, closed\}$
  actions
    open close
  axioms
    perm $open \rightarrow state = closed$
    $[open]\ state = open$
    perm $close \rightarrow state = open$
    $[close]\ state = closed$

Fig. 2. The space interactor

interactor *spaceSign*
  aggregates
    window **via** *oI*
    window **via** *cI*
  attributes
    $\boxed{\text{vis}}$ $state : \{open, closed\}$
  actions
    $\boxed{\text{vis}}$ *open close*
  axioms
    perm $open \rightarrow state = closed$
    $[open]\ state' = open$
    perm $close \rightarrow state = open$
    $[close]\ state' = closed$
    only $open\ oI.update\ oI.show$ $\vee$ only $close\ cI.update\ cI.show$

Fig. 3. A classical solution

composed in the 'classical' way, *i.e.*, by a specification *import* mechanism, illustrated below by means of a small example. This will be contrasted in section 5 to a coordination-based solution. Consider a system that controls access to a specific space (e.g, an elevator), modelled by the interactor in Fig. 2. Now suppose two indicators have to be added to this model, one to announce *open* events, the other to signal *close* events. We will use instances of the window interactor from Fig. 1 to act as indicators. The 'classical' aggregation strategy, as in [8], requires that two instances of the *window* interactor be imported into one instance of *space* to build the new interactor. The rules that govern their incorporation are as follows:

- the *open* (respectively, *close*) indicator must be made visible and have its information updated whenever the system is opened (respectively, closed).

Additionally, it should be noted that whenever a window is made visible, it might overlap (and hide) another one. The resulting interactor is presented in figure

3, where a new axiom expresses the coordination logic. The fact that $\mathbb{M}$ allows for action co-occurrence means that constraints on actions become simpler and more concise than their MAL counterparts, as used in [8]: in our example only an additional axiom is needed. Nevertheless, this solution still mixes concerns by expressing the coordination of interactors *cI* and *oI* at the same level than the internal properties of the underlying *space* interactor. How such two levels can be disentangled is the topic of the following sections.

4 The Coordination Layer

Actually, coordination entails a different perspective. As in [1] this is achieved through specific *connectors* which abstract the idea of an intermediate *glue code* to handle interaction. Connectors have *ports*, thought of as *interface points* through which messages flow. Each port has an *interaction polarity* (either *input* or *output*), but, in general, connectors are blind with respect to the data values flowing through them. The set of elementary interactions of a connector $\mathbb{C}$ forms its *sort*, denoted by $\mathsf{sort}.[\![\mathbb{C}]\!]$. By default the sort of $\mathbb{C}$ is the set of its ports, but often such is not the case. For example, a synchronous channel with ports a and a' has a unique possible interaction: the simultaneous activation of both a and a', represented by aa'.

Connectors are specified at two levels: the *data* level, which records the flow of data, and the *behavioural* one which prescribes all the activation patterns for ports. Formally, let $\mathbb{C}$ be a connector with m input and n output ports. Assume $\mathbb{D}$ as a generic type of data values and $\mathbb{P}$ as a set of (unique) *port identifiers*. Then,

Definition 2. *The specification of a connector* $\mathbb{C}$ *is given by a relation* $\mathsf{data}.[\![\mathbb{C}]\!] : \mathbb{D}^n \longleftarrow \mathbb{D}^m$*, which relates data present at its* m *input ports with data at its* n *output ports, and an* $\mathbb{M}$ *assertion,* $\mathsf{port}.[\![\mathbb{C}]\!]$*, over its sort,* $\mathsf{sort}.[\![\mathbb{C}]\!]$*, which specifies the relevant properties of its port activation pattern.*

4.1 Elementary Connectors

The most basic connector is the *synchronous channel* which exhibits two ports, a and a', of opposite polarity. This connector forces input and output to become mutually blocking. Formally, $\mathsf{data}.[\![\ a \longrightarrow a'\ ]\!] \ = \ \mathsf{Id}_{\mathbb{D}}$, i.e., the identity relation in $\mathbb{D}$, and

$$\mathsf{sort}.[\![\ a \longrightarrow a'\ ]\!] \ = \ \{aa'\} \quad \mathsf{port}.[\![\ a \longrightarrow a'\ ]\!] \ = \ \mathsf{only}\, aa'$$

Its static semantics is simply the identity relation on data domain $\mathbb{D}$ and its behaviour is captured by the simultaneous activation of its two ports.

Any coreflexive relation provides channels which can loose information, thus modelling unreliable communications. Therefore, we define, an *unreliable channel* as $\mathsf{data}.[\![\ a \xrightarrow{\diamond} a'\ ]\!] \ \subseteq \ \mathsf{Id}_{\mathbb{D}}$ and

$$\mathsf{sort}.[\![\ a \xrightarrow{\diamond} a'\ ]\!] \ = \ \{a, aa'\} \quad \mathsf{port}.[\![\ a \xrightarrow{\diamond} a'\ ]\!] \ = \mathsf{only}\, a$$

The behaviour expression states that all valid transitions involve input port a, although not necessarily a'. This corresponds either to a successful communication, represented by the simultaneous activation of both ports, or to a failure, represented by the single activation of the input port.

As an example of a connector which is not stateless consider *fifo*$_1$, a channel with a buffer of a single position. Formally, $\mathsf{data}.[\![\ a \longrightarrow\!\square\!\!\rightarrow a'\ ]\!] \ = \ \mathsf{Id}_{\mathbb{D}}$ and

$$\mathsf{sort}.[\![\ a \longrightarrow\!\square\!\!\rightarrow a'\ ]\!] \ = \ \{a, a'\} \quad \mathsf{port}.[\![\ a \longrightarrow\!\square\!\!\rightarrow a'\ ]\!] \ = \ [a]\mathsf{only}\, a', \sim\! a$$

Notice that its port specification equivales to $[a](\mathsf{only}\, a' \wedge \mathsf{forbid}\, a)$, formalising the intuition of a strict alternation between the activation of ports a and a'.

If channels forward information, *drains* absorb it. However they play a fundamental role in controlling the flow of data along the coordination code. A *drain* has two input, but no output, ports. Therefore, it looses any data item crossing its boundaries. A drain is *synchronous* if both write operations are requested to succeed at the same time (which implies that each write attempt remains pending until another write occurs in the other end-point). It is *asynchronous* if, on the other hand, write operations in the two ports do not coincide. The data part coincides in both connectors: $\mathbb{D} \times \mathbb{D}$. Then

$$\mathsf{sort}.[\![\ a \overset{\blacktriangledown}{\longmapsto\!\dashv} a'\ ]\!] \ = \ \{aa'\} \quad \mathsf{port}.[\![\ a \overset{\blacktriangledown}{\longmapsto\!\dashv} a'\ ]\!] = \mathsf{only}\, aa'$$

$$\mathsf{sort}.[\![\ a \overset{\triangledown}{\longmapsto\!\dashv} a'\ ]\!] \ = \ \{a, a'\} \quad \mathsf{port}.[\![\ a \overset{\triangledown}{\longmapsto\!\dashv} a'\ ]\!] = \mathsf{only}\, a, a' \ \wedge \ \mathsf{forbid}\, aa'$$

4.2 New Connectors from Old

Connectors can be combined in three different ways: by placing them side-by-side, by sharing ports or introducing feedback wires to connect output to input ports. In the sequel, note that behaviour annotations in the specification of connectors can always be presented in a *disjunctive* form

$$\mathsf{port}.[\![\mathbb{C}]\!] \ = \ \phi_1 \vee \phi_2 \vee \cdots \vee \phi_n \tag{7}$$

where each ϕ_i is a conjunction of

$$\underbrace{[K]\cdots[K]}_{n}\ \mathsf{only}\, F$$

Also let $t_{|a}$ and $t_{\#a}$, for $t \in \mathbb{D}^n$ and $a \in \mathcal{P}$, represent, respectively, a tuple of data values t from which the data corresponding to port a has been deleted, and the tuple component corresponding to such data. Then,

Join. This combinator places its arguments side-by-side, with no direct interaction between them. Then,

$$\begin{aligned}
\mathsf{data}.[\![\mathbb{C}_1 \ \boxtimes \ \mathbb{C}_2]\!] \ &= \mathsf{data}.[\![\mathbb{C}_1]\!] \ \times \ \mathsf{data}.[\![\mathbb{C}_2]\!] \\
\mathsf{sort}.[\![\mathbb{C}_1 \ \boxtimes \ \mathbb{C}_2]\!] \ &= \mathsf{sort}.[\![\mathbb{C}_1]\!] \ \cup \ \mathsf{sort}.[\![\mathbb{C}_2]\!] \\
\mathsf{port}.[\![\mathbb{C}_1 \ \boxtimes \ \mathbb{C}_2]\!] \ &= \mathsf{port}.[\![\mathbb{C}_1]\!] \ \vee \ \mathsf{port}.[\![\mathbb{C}_2]\!]
\end{aligned}$$

The relevance of sorts becomes now clear. Take, for example, the aggregation of two synchronous channels Their joint behaviour is

$$\mathsf{port}.[\![(\ a \longrightarrow a' \ \boxtimes \ c \longrightarrow c' \ )]\!] \ = \ \mathsf{only}\, aa' \ \vee \ \mathsf{only}\, cc'$$

A transition labelled by, say, $aa'c$ does not violate the behaviour prescribed above, but it is made invalid by the sort specification, which is $\{aa', cc'\}$.

Share. The effect of *share* is to plug ports with identical polarity. The aggregation of *output* ports is done by a *right share* ($\mathbb{C}\ {}^{i}_{j} > z$), where $\mathbb{C}$ is a connector, i and j are ports and z is a fresh name used to identify the new port. Port z receives asynchronously messages sent by either i or j. When input from both ports is received at same time the combinator chooses one of them in a non-deterministic way. Let $\mathsf{data}.[\![\mathbb{C}]\!] : \mathbb{D}^n \longleftarrow \mathbb{D}^m$. Then, the data flow relation $\mathsf{data}.[\![\mathbb{C}\ {}^{i}_{j} > z]\!] : \mathbb{D}^{n-1} \longleftarrow \mathbb{D}^m$ for this operator is given by

$$r\,(\mathsf{data}.[\![\mathbb{C}\ {}^{i}_{j} > z]\!])\,t \ \Leftrightarrow\ t'\,(\mathsf{data}.[\![\mathbb{C}]\!])\,t \ \ \wedge\ \ r_{|z} = t'_{|i,j} \ \wedge\ (r_{\#z} = t'_{\#i} \ \vee\ r_{\#z} = t'_{\#j})$$

At the behavioural level, its effect is that of a renaming applied to the $\mathbb{M}$-formula capturing the behavioural patterns of $\mathbb{C}$, *i.e.*,

$$\mathsf{port}.[\![(\mathbb{C}\ {}^{i}_{j} > z)]\!] \ = \ \{z \leftarrow i, z \leftarrow j\}\ \mathsf{port}.[\![\mathbb{C}]\!]$$

over

$$\mathsf{sort}.[\![(\mathbb{C}\ {}^{i}_{j} > z)]\!] \ = \ \{z \leftarrow i, z \leftarrow j\}\ \mathsf{sort}.[\![\mathbb{C}]\!]$$

Figure 4 represents a *merger* formed by sharing the output ports of a synchronous channel and a 1-place buffer.

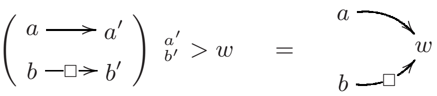


Fig. 4. A *merger*: $\mathsf{only}\, aw \vee [b]\mathsf{only}\, w, \sim\! b$

On the other hand, aggregation of *input* ports is achieved by a *left share* mechanism ($z <^{i}_{j} \mathbb{C}$). This behaves like a *broadcaster* sending synchronously messages from z to both i and j. This case is slightly more complex: before renaming, all computations of $\mathbb{C}$ in which ports i and j are activated independently of each other must be synchronised. Therefore, we take all disjuncts in $\mathsf{port}.[\![\mathbb{C}]\!]$ in which ports i and j are involved, form their conjunction to force co-occurrence, and apply renaming. Formally, let ϕ_θ be a disjunct in $\mathsf{port}.[\![\mathbb{C}]\!]$ (recall (7)) involving only ports in θ. Define $\phi_i = \langle \bigvee\ \phi_\theta \in \mathsf{port}.[\![\mathbb{C}]\!] \ :\ i \in \theta :\ \phi_\theta \rangle$ and, similarly, ϕ_j. Therefore, for $\sigma = \{z \leftarrow i, z \leftarrow j\}$,

$$\mathsf{port}.[\![(z <^{i}_{j} \mathbb{C})]\!] \ = \ \sigma(\phi_i \wedge \phi_j) \ \vee\ \langle \bigvee\ \phi_{\theta'} \in \mathsf{port}.[\![\mathbb{C}]\!] \ :\ i \notin \theta' \wedge j \notin \theta' :\ \phi_{\theta'} \rangle$$

and, again,

$$\mathsf{sort}.[\![(z <^i_j \mathbb{C})]\!] \;=\; \{z \leftarrow i, z \leftarrow j\}\, \mathsf{sort}.[\![\mathbb{C}]\!]$$

On the other hand, relation $\mathsf{data}.[\![z <^i_j \mathbb{C}]\!] : \mathbb{D}^n \longleftarrow \mathbb{D}^{m-1}$ is given by

$$t' \,(\mathsf{data}.[\![z <^i_j \mathbb{C}]\!])\, r \;\leftrightarrow\; t' \,(\mathsf{data}.[\![\mathbb{C}]\!])\, t \;\wedge\; r_{|z} = t_{|i,j} \;\wedge\; r_{\#z} = t_{\#i} = t_{\#j}$$

As an example let us calculate the sharing of input ports a and b in a connector composed by three, otherwise non interfering, synchronous channels,

$$\begin{array}{ll}
 & \mathsf{port}.[\![z <^a_b (\; a \longrightarrow a' \;\boxtimes\; b \longrightarrow b' \;\boxtimes\; c \longrightarrow c' \;)]\!] \\
\Leftrightarrow & \quad \{ \text{ definition } \} \\
 & \{z \leftarrow a, z \leftarrow b\}(\mathsf{only}\, aa' \wedge \mathsf{only}\, bb') \;\vee\; \mathsf{only}\, cc' \\
\Leftrightarrow & \quad \{ \text{ renaming and (3) } \} \\
 & \mathsf{only}\, za', zb' \;\vee\; \mathsf{only}\, cc' \\
\Leftrightarrow & \quad \{ \text{ (5) } \} \\
 & \mathsf{only}\, za'b' \;\vee\; \mathsf{only}\, cc'
\end{array}$$

which asserts that input on z co-occurs with output at both a' and b'. Replacing $b \longrightarrow b'$ by a one-place buffer leads to the connector depicted in Fig. 5 which is calculated as follows

$$\begin{array}{ll}
 & \mathsf{port}.[\![z <^a_b (\; a \longrightarrow a' \;\boxtimes\; b \xrightarrow{\square} b' \;\boxtimes\; c \longrightarrow c' \;)]\!] \\
\equiv & \quad \{ \text{ definition } \} \\
 & \{z \leftarrow a, z \leftarrow b\}(\mathsf{only}\, aa' \wedge [b]\mathsf{only}\, b', {\sim}b) \;\vee\; \mathsf{only}\, cc' \\
\equiv & \quad \{ \text{ renaming } \} \\
 & (\mathsf{only}\, za' \wedge [z]\mathsf{only}\, b', {\sim}b) \;\vee\; \mathsf{only}\, cc'
\end{array}$$

Hook. This combinator encodes a *feedback* mechanism, drawing a direct connection between an output and an input port. This has a double consequence: the connected ports must be activated simultaneously and become externally non observable. The formal definition is omitted here (but see [5]) because this combinator is not used in the examples to follow. For the sake of curiosity, note

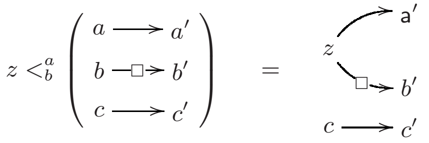


Fig. 5. A *broadcaster* and a detached channel

the following 'extreme' situations arising from hooking a synchronous channel and a 1-place buffer, respectively,

$$(\mathsf{only}\, aa')\ \urcorner^{a}_{a'} \;=\; \mathsf{only}\,\emptyset \;=\; \mathsf{true}$$

$$[a](\mathsf{only}\, a', {\sim}a)\ \urcorner^{a}_{a'} \;=\; [\emptyset](\mathsf{true} \wedge \mathsf{false}) \;=\; \mathsf{false}$$

as one may have expected given the buffer strict alternation activation discipline.

5 Configurations of $\mathbb{M}$-Interactors

Having introduced $\mathbb{M}$-interactors and the coordination layer on top of the same modal language, we may now complete the whole picture. The key notion is that of a *configuration*, *i.e.*, a collection of *interactors* interconnected through a *connector* built from elementary connectors, combined trough the combinators defined above. Formally,

Definition 3. *A configuration is a tuple $\langle I, \mathbb{C}, \sigma\rangle$, where $I = \{I_i |\ i \in n\}$ is a collection of interactors, $\mathbb{C}$ is a connector and σ a mapping of ports in I to ports in $\mathbb{C}$. The behaviour of a configuration is given by the conjunction of the modal theories for each $I_n \in I$, as specified by their axioms, and the port specification* $\mathsf{port}.[\![\mathbb{C}]\!]$ *of connector $\mathbb{C}$, after renaming by σ.*

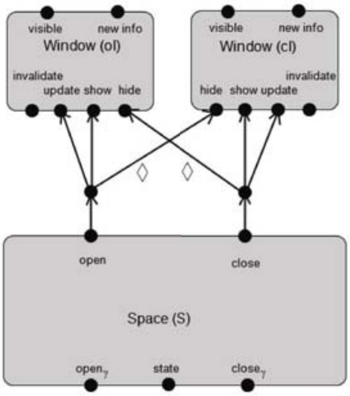


Fig. 6. A coordination-based solutio

To illustrate the envisaged approach, consider again the example discussed in section 3. A coordination-based solution, depicted in Fig. 6, replaces the hierarchical import of *window* into *spaceSign* interactor, by a configuration in which the two instances of the former and one instance of the original *space* interactor are connected by

$$\mathbb{BC} \;\triangleq\; \mathbb{B} \boxtimes \mathbb{B}$$

a connector which joins together two broadcasters $\mathbb{B}$. Each $\mathbb{B}$ is formed by two synchronous channels and a lossy channel, sharing their input ports, *i.e.*

$$\mathbb{B} \;\triangleq\; z <^{w}_{c} (w <^{a}_{b} (\ a \longrightarrow a' \ \boxtimes\ b \longrightarrow b') \ \boxtimes\ c \xrightarrow{\ \diamond\ } c')$$

An easy calculation yields $\mathsf{port}.[\![\mathbb{B}]\!] \;=\; \mathsf{only}\, za', zb', z$, which, by (5), equivales to $\mathsf{only}\, za'b'$. In a configuration in which, through a renaming σ, port z is linked to *S.open*, a' to *oI.update*, b' to *oI.show* and c' to *cI.hide*, one may prove (*i.e.*, discover, rather than assert) a number of desirable properties of the configuration. For example, from axiom $\mathsf{perm}\, S.open$, a default axiom of interactor *space* in section 3, and $\sigma\, \mathsf{only}\, za'b'$, one concludes that

$$\mathsf{perm}\, S.open\ oI.update\ oI.show$$

i.e., there are transitions in which all the three ports are activated at the same time. But, because the connector does not allow actions not including the simultaneous activation of such three ports, the joint behaviour of the configuration asserts not only possibility but also necessity of this transition, *i.e.*,

$$\mathsf{perm}\, S.open\ oI.update\ oI.show\ \wedge\ \mathsf{only}\, S.open\ oI.update\ oI.show$$

This is stronger than the corresponding axiom added to interactor *spaceSign* in Fig. 3, although it can be deduced from the modal theory of this interactor (which, of course, includes $\mathsf{perm}\, open$). Note we are focussing only on one of the two $\mathbb{B}$ connectors in $\mathbb{BC}$, thus this conclusion does not interfere with a similar possibility for the other connection of interactor instances S and cI (recall the behavioural effect of $\boxtimes$ is disjunction).

On the other hand, one also has $\mathsf{perm}\, S.open\, cI.hide$ because action zc' is in $\mathsf{sort}.[\![\mathbb{B}]\!]$, but, now only as a possibility, because an unreliable channel was used to connect these ports. From this property and $\mathsf{only}\, S.open\, oI.update\, oI.show$ above, we can easily conclude that $cI.hide$ cannot occur independently of $S.open$, $oI.update$ and $oI.show$. Again, this is stronger than the interactor model in Fig. 3, where the hide action was left unrestricted.

As a final example, consider an interactor which has to receive the location coordinates supplied by two different input devices but in strict alternation. The connector to plug these three interactors is the *alternate merger* depicted in Fig. 7, formally, defined as

$$b <^{d'}_{f} a <^{d}_{c} (\ c \longrightarrow c'\ \boxtimes\ d \xrightarrow{\blacktriangledown} d'\ \boxtimes\ f \xrightarrow{\square} f')^{c'}_{f'} > w$$

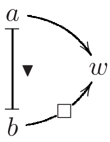


Fig. 7. An *alternate merger*

Its behavioural pattern is

$$\mathsf{port}.[\![\mathbb{AM}]\!]\ =\ \mathsf{only}\, awb\ \wedge\ [b]\mathsf{only}\, w, \sim b$$

Clearly, each activation of port a is synchronous with b and w. But then data received in b (say, the coordinates of the one of the devices) is stored in the buffer. Next action is necessarily w, whose completion empties the buffer.

6 Conclusions and Future Work

It was our intention to set the foundations for an approach to modelling interactive systems entailing a true separation of concerns between modelling of individual components (interactors) and their architectural organisation. For this a new modal logic (the $\mathbb{M}$ language) was introduced, which is similar to the Hennessy-Milner logic [14] but for the fact that its modal connectives are indexed by sets of actions (*actions factors*). These action factors are interpreted over the compound actions (themselves represented by sets) that label transitions using set inclusion. This makes it possible to express properties over co-occurring actions in the logic.

Although the main drive behind the development of $\mathbb{M}$ was the need for a modal logic expressive enough to define the coordination layer, the language was also used to specify interactors, thus providing a single language for expressing the behaviour of both interactors and connectors that bind them.

The approach presents two major benefits over [13] or [8]. First of all, it promotes a clear separation of concerns between the specification of the individual interactors and the specification of how they interact with each other. Furthermore, it frees us from the rigid structure imposed by hierarchical organisation.

At this point, it is worthwhile pointing out that when composing interactors into different configurations, the resulting behaviour becomes an emergent feature of the model. Hence, we discover, rather than assert, what the system will be like. This is particularly relevant in a context were one is interested in exploring the impact of different design decisions at the architectural level. Recent related work on the use of (some type of) logic to specify component behaviour include [7] and [15], the latter with an emphasis on property verification.

A number of lines of research have been opened by the current endeavour. A main one concerns temporal extension. Actually, language $\mathbb{M}$ seems expressive enough to express connector's behaviour, but not so when facing more elaborate interactor's specifications. A typical case relates to expressing *obligation* requirements. We are currently studying how $\mathbb{M}$ can be extended in a way similar to D. Kozen's μ-calculus [16] in order to address these temporal issues.

Acknowledgements. The authors wish to thank Michael D. Harrison for useful comments on a preliminary version of this paper.

References

1. Arbab, F.: Reo: a channel–based coordination model for component composition. Mathematical Structures in Comp. Sci. 14(3), 329–366 (2004)
2. Barbosa, M.A., Barbosa, L.S.: A relational model for component interconnection. Journal of Universal Computer Science 10(7), 808–823 (2004)
3. Barbosa, M.A., Barbosa, L.S., Campos, J.C.: Towards a coordination model for interactive systems. In: Cerone, A., Curzon, P. (eds.) FMIS 2007: Proc. 1st Inter. Workshop in Formal Methods for Interactive Systems. Electronic Notes in Theoretical Computer Science, vol. 347, pp. 89–103. Elsevier, Amsterdam (2007)
4. Barbosa, M.A., Barbosa, L.S.: Specifying software connectors. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 52–67. Springer, Heidelberg (2005)
5. Barbosa, M.A.: Specification and Refinement of Software Connectors. PhD thesis, DI, Universidade do Minho (to appear, 2009)

6. Bastide, R., Navarre, D., Palanque, P.A.: A tool-supported design framework for safety critical interactive systems. Interacting with Computers 15(3), 309–328 (2003)
7. Bowles, J.K.F., Moschoyiannis, S.: Concurrent logic and automata combined: A semantics for components. In: Canal, C., Viroli, M. (eds.) Proc. of FOCLASA 2006, vol. 175(2), pp. 135–151. Elsevier, Amsterdam (2006)
8. Campos, J.C., Harrison, M.D.: Model checking interactor specifications. Automated Software Engineering 8(3/4), 275–310 (2001)
9. Campos, J.C., Harrison, M.D.: Systematic analysis of control panel interfaces using formal tools. In: Graham, T.C.N., Palanque, P. (eds.) DSV-IS 2008. LNCS, vol. 5136, pp. 72–85. Springer, Heidelberg (2008)
10. d'Ausbourg, B., Seguin, C., Durrieu, G., Roché, P.: Helping the automated validation process of user interfaces systems. In: ICSE 1998: Proc. 20th Inter. Conf. on Software Engineering, pp. 219–228. IEEE Computer Society, Los Alamitos (1998)
11. Dittmar, A., Forbrig, P.: A unified description formalism for complex hci-systems. In: SEFM 2005: Proc. 3rd IEEE Inter. Conf. on Software Engineering and Formal Methods, pp. 342–351. IEEE Computer Society, Los Alamitos (2005)
12. Duke, D.J., Harrison, M.D.: Abstract interaction objects. Computer Graphics Forum 12(3), 25–36 (1993)
13. Faconti, G., PaternÚ, F.: An approach to the formal specification of the components of an interaction. In: Vandoni, C., Duce, D. (eds.) Eurographics 1990, pp. 481–494. North-Holland, Amsterdam (1990)
14. Hennessy, M.C., Milner, A.J.R.G.: Algebraic laws for non-determinism and concurrency. Journal of ACM 32(1), 137–161 (1985)
15. Johnsen, E.B., Owe, O., Torjusen, A.B.: Validating behavioural component interfaces in rewriting logic, vol. 159, pp. 187–204. Elsevier, Amsterdam (2006)
16. Kozen, D.: Results on the propositional μ-calculus. Theor. Comp. Sci. (27), 333–354 (1983)
17. Markopoulos, P.: On the expression of interaction properties within an interactor model. In: Palanque, P., Bastide, R. (eds.) Design, Specification and Verification of Interactive Systems 1995 (1995)
18. PaternÜ, F.D.: A Method for Formal Specification and Verification of Interactive Systems. PhD thesis, Department of Computer Science, University of York (1995); Available as Technical Report YCST 96/03
19. Ryan, M., Fiadeiro, J., Maibaum, T.: Sharing actions and attributes in Modal Action Logic. In: Ito, T., Meyer, A.R. (eds.) TACS 1991. LNCS, vol. 526, pp. 569–593. Springer, Heidelberg (1991)

Evolution Control in MDE Projects: Controlling Model and Code Co-evolution

Jacky Estublier, Thomas Leveque, and German Vega

LIG-IMAG, 220, rue de la Chimie BP53, 38041 Grenoble Cedex 9, France
{Jacky.Estublier,Thomas.Leveque,German.Vega}@imag.fr

Abstract. The dream of Model Driven Engineering (MDE) is that Software Engineering activities should be performed only on models, but in practice a significant amount of programming is still being performed. There is a clear need to keep code and models strongly synchronized when they represent the same entities at different levels of abstraction. We observe that versioning is ill supported by MDE tools, and that no strong synchronization is ensured between code and model versions. This, among other things, explains why MDE is not widely adopted in industry.

This paper presents the solution developed in the CADSE project for providing consistent support for model and code co-evolution. It is shown that it requires to (1) define, what evolution policy is to be applied, (2) closely synchronize both ways, the model entities and the computer artifacts, and (3) enforce consistency constraints and evolution policies during the commit and check-out of both model elements and their corresponding artifacts.

Keywords: Domain Specific Languages (DSL), metamodel-based environments, composition of DSL interpreters, composition of models.

1 Introduction

The main claim of Model Driven Engineering (MDE) [9] is that Software Engineering activities will be performed mostly, if not only, on models, all along the life cycle of software products. Almost 10 years later, this vision is still not a reality in industry, except for a few niches and in certain specific conditions [2].

It is an observation that the major progresses in Software Engineering (code engineering) are due to the high quality of environments and tools that assist Software Engineers in their day to day work. A major problem faced, and to a large extent solved, is evolution control; but it took three decades from SCCS [8] before obtaining reliable and convenient version and configuration control systems.

No such history is available for MDE. Today, model versioning and merging is a current research topic in its infancy. Therefore, no evolution-control system of industrial strength is currently available for model engineering. What makes the situation even worse is that in practice, a large fraction of Software Engineering

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 431–438, 2010.

activities consists in developing code and many other files, such as scripts, metadata, or documentation. From a purist MDE point of view, all these files are models, but from a Software Engineering perspective, these files are managed in the "old" way, relying on traditional evolution control tools, while no such tool exist for (real) models.

This situation requires the definition of new evolution control paradigms for software projects that are made of a mixture of models and files. On one hand, these evolution paradigms must take into account the nature of models and must be adapted to model engineering practices. On the other hand, such paradigms must respect the code-engineering practice and must rely on the tools and systems available in traditional Software Engineering. We believe that this lack of consistent evolution support, on all grounds - policies, methods, tools, and environment, is one of the major reasons why MDE did not succeeded in industry so far.

This paper shows that at least three issues have to be addressed for defining such evolution paradigm and its associated tools. We believe that evolution control, in MDE projects, requires solving the three issues:

- Synchronization of model elements with code and files
- Definition and support of evolution policies
- Definition and support of consistency constraints

Section 2 presents how are synchronized model elements and code, section 3 describes how can be defined evolution policies, and section 4 shows how consistency is computed and enforced. Finally, section 5 concludes and proposes future works.

2 Synchronizing Model and Software Artifacts

Model transformations began to be used for maintaining consistency between model and software artifacts when the application code is fully derived from a model [5]. In that case, users only work on the model, while artifacts are (re)generated when needed; in other words, the model is a high-level source code.

In general, models do not contain enough details to be executable. This is why developers work on model and code at the same time. Usually, code skeletons are generated from the model, but the model cannot be reconstructed from artifacts and vice versa. Hence, we fall into synchronization issues where modifications on model and artifacts must be reconciled. Few MDE tools support permanent synchronization both ways between model and artifacts.

In our CADSE environment [4], traceability links can be defined between the model and the artifacts. These links translate the operations performed on the model to modifications performed on artifacts and vice-versa. It is possible, for example, to define that the concept of *service* defined in a metamodel should be mapped to an Eclipse java project with a specific structure and specific files (e.g. metadata information and templates). The synchronization ensures that each

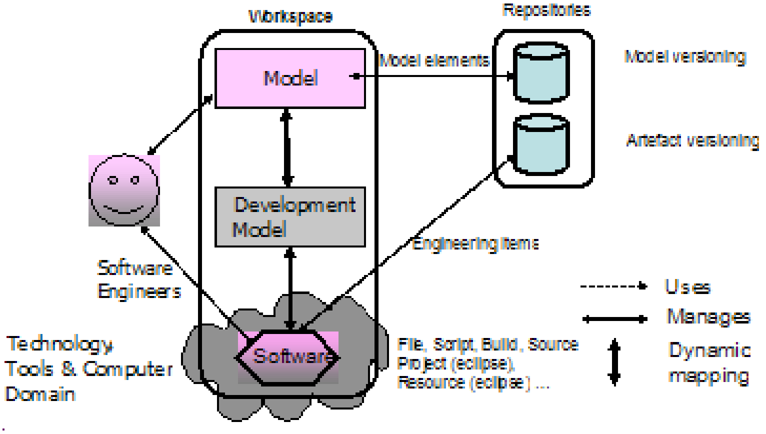


Fig. 1. Model and code versioning

time a *service* is defined in a model the corresponding Eclipse project is created. Conversely, changes in some files (metadata) are translated into attributes and relationships in the model.

This synchronization mechanism enables Software Engineering activities be performed either on models or on files, depending on the nature of the activity, while enforcing consistency between both views. This means that a model element and its mapping must evolve in concert, and since versioning is defined at the file granularity, versioning must also be applied on each model element individually. This characteristic alone disqualifies most model versioning approaches where the grain of versioning is a complete model, or a large fraction of it.

The CADSE environment is an eclipse plug-in. The model is found in an Eclipse window, while the artefacts are provided as eclipses projects or files. Under Eclipse, users can directly create, delete, or change model elements and execute engineering operations such as build, package, or edit on these elements.

3 Evolution Policies

Versioning should be a mechanism that serves an evolution policy. Unfortunately, in most systems, only the mechanism exists, while the evolution policy remains undefined and relies on the good will of developers. In the CADSE project, the ambition is to make explicit and automatic the evolution policy. To that end, we have defined a version model and a versioning mechanism and we show how versioning policies can be defined and enforced.

3.1 Data Model, Version Model

We use ECore [1], which is an Object oriented data model. Basic entities are classes, which have attributes and operations, and which are related to other classes by relations and possibly by the special inheritance relationship. This is enough for our purpose (please refer to [3] for more details), but this data model does not include any versioning concept.

Before addressing evolution policies, we need to define the concept of a version and its associated *version model*. Two objects are versions of each other if they share something (otherwise they are two independent objects) and if they are different in some way (otherwise it is the same object). Following [6], we decided to make explicit the common part constituted of shared attributes, and the difference as two objects. The different parts (not common one) are called "the revisions". A versioned object (by language abuse often called a revision) is the union of the information found in the common part and the one found in one of its revisions. A branch is the common part and all its revisions. By convention, the name of a branch is the name of the common part. This is a symbolic name, where the name of a revision is an integer and the name of a versioned object has the form "*branch.revision*".

3.2 Object Evolution Control

Evolution control refers to the criteria by which new versions of an object are created. Two classes of criteria can be identified: (1) how important is the change that has been made on an object (to which extend it changes its semantics); and (2) when to record the change. In theory, a change should be recorded only when the object is in a stable state. Unfortunately, we do know enough on the object semantics to evaluate the object state which is mostly on the artifact state: is the code of videoPlayer.3 consistent, reliable and so on. For that reason, as usual, the decision to commit changes is left to the engineer. Depending on the change importance, versioning may mean:

- Update the current value of the object in the repository
- Create a new revision of the object (in same variant)
- Create a new variant
- Create a new object

Traditionally, this choice is also left to the user. However the importance of a change can be related to the semantics embedded in each attribute.

In our system, a versioning characteristic which can be *mutable, immutable, final* or *transient* is associated to each attribute and relationship. If a *mutable* attribute is changed, the next commit will simply update its value in the repository. If an *immutable* and *revision* attribute is changed, the next commit will create a new revision of the object. If an *immutable* and *shared* attribute is changed, next commit will create a new branch. Changes in a *transient* attribute are supposed to be transparent for the versioning system and *final* attributes, if they need to be changed, must create a new object.

3.3 Relationship Evolution Control

Relationships are seen as attributes of the source element and can consequently have the *mutable*, *immutable*, *final, transient* evolution characteristics. However, due to our version model, the origin of a relation can be a revision, or a branch, and the destination can also be a branch, or a destination. This leads to the definition of different kinds of relationships:

– Global references (Branch to Branch). Only the last state (entity source and entity destination) of the link is saved. This is called product first in [7].
– Contextual references (Revision to branch). This is the way to define a non-versioned entity.
– Branch to revision. This is possible, but seldom used.
– Version-specific references (Revision to revision).
– Effective references (X to Set(Revision)). Instead of pointing on one or all revisions of an entity, effectivity, which comes from the PDM world [3], associates a subset of all revisions.

In section 3.2, we have shown that each attribute and relationship can be annotated to indicate how the object should react (from the versioning point of view) when that attribute changes. Based on attribute annotations, we can update, create a new revision or create a new variant only when needed. But this mechanism ignores the impact of such changes on the surrounding objects. This is what we call change propagation, defined at type level, using the following annotations on relationships:

– **MutableDestination**: changing the revision number of the destination object does not have impact on the origin entity, but now the link targets the new one.
– **ImmutableDestination**: a new revision of the origin entity is created when the destination object revision changes.
– **EffectiveDestination**: the entity is compatible with the previous and the new destination state.
– **FinalDestination**: the destination is not allowed to be modified.

The annotations on attributes and link allow defining (at type level) the evolution strategy to be applied. It allows, for each kind of change, for the precise definition of the evolution control that the system should perform. It also allows the definition of the versions to be created (i.e. update, revision, or variant), and the objects to which the change propagates.

Our objective is to avoid version proliferation, which is why defining state and action propagation is a priority for us. In all cases, it is important to realize that these annotations are defined at type level by software experts. In a workspace, the developers only "see" a versioned object, the distinction branch/revision and the revision numbers are not visible. Additionally, propagation and evolution strategy are fully automated and transparent.

4 Definition and Support of Consistency Constraints

Remember that the goal of versioning is to store objects in a given state so that they can be used at a later time. One purpose of storing different versions is to recover from crashes and mistakes, but the major purpose is to reuse stored versions in different assemblies. Reuse is the major driver for setting versioning strategies.

Optimizing reuse entails a number of issues. In theory, the lower is the granularity of the objects to reuse, the higher is the number of possible different assemblies. This is theoretical only because: (1) not all assemblies are valid, and (2) the capability for a human to build a complete and consistent assembly without help is limited. Indeed, these reasons explain the current practices. As they are not able to build an assembly piece by piece, developers only use existing assemblies (snapshots, baselines), built and checked by somebody else. Any change in the assembly produces a new revision of everything. Consequently, this results in a huge number of useless revisions, as only the new revision of the baseline will actually be used. Ironically, this practice generates a huge number of revisions of everything, while in practice the grain of reuse is highly coarse. Namely, the reuse granularity is the complete baseline which explains why reuse is far from optimal. Optimizing reuse requires the following:

- Allow all possible assemblies.
- Avoid inconsistent assemblies.
- Help in building complete and consistent assemblies.

From a versioning point of view, these requirements translate as: (1) a version should be at the lowest possible granularity level, (2) version compatibility should be defined and controlled, (3) the number of versions should be minimized and (4) the dependencies should be known.

The propagation control mechanism described above already solves the first and the last of the above requirements. This is because the granularity is the element, because the minimum side effects of a change are computed (relationship propagation) in order to find out the elements that really need to be saved (modified) and because revisions are only created when required (immutable attributes). All and only the useful versions (updates, revisions and variants) are created.

For two elements, *o.i* and *d.j*, ***link r is said to be consistent*** if *o* or *o.i* is the origin of *r*, and if *d* is the destination (if *r* is branch), or if *j* pertains to the effectivity of *r*. ***An assembly is said to be consistent***, if all links between elements pertaining to the assembly are consistent. We define a ***required-set*** as a set of relationship types RS=R1, . . . Rn. ***An assembly is said to be complete***, with respect to a required-set RS, if for all elements of the assembly, each link whose type is in RS leads to another element of the same assembly.

Note that the above definition of consistency relies on the effectivity mechanism. On one hand, the definition is conservative because the system only records in the effectivity those pairs (origin-destination) that have been tested in a workspace. Other pairs may be consistent but have not been tested. On the other hand, the effectivity is not a formal proof of compatibility. More precisely, the system assumes that only assemblies that have been tested are committed on the shared repository.

Workspace management is strongly linked to these definitions. In our system, assemblies can be explicitly built, by importing the element *d* or revision *d.j* that the user thinks is needed from the shared repository to the workspace. But before importing *d*, the system checks which revisions of *d* are consistent with the elements already present in the current workspace. If such revisions *d* exist, then the system imports the most recent revision of d. If no such revision of d exists the system sends a warning to the user. This manual process can be widely facilitated by declaring some relationship type as *required.* In this case, importing an element automatically imports all *required* elements. Associating a required-set RS to a workspace is such that importing an element ensures the completeness of that element with respect to RS.

Based on these mechanisms, creating, maintaining and evolving complete and consistent assemblies is easy and does not require from the user any knowledge of dependencies and compatibility.

5 Conclusion

The MDE approach is still novel and as such lacks methods and tools required by practitioners. This work seeks to provide concepts, methods and tools that allow for closely controlling the evolution of a software product, all along its life cycle, while following a MDE approach. First, developers have to deal with both models and files. These files could represent programs in case models are not rich enough to be executable, which is most often the case, and in case those files are needed for compiling, packaging, or documenting. We believe that this situation is not due to the transition from code to model engineering but that, at least for the foreseeable future, models and files will co-exist and will have to be managed together consistently.

The models we suggest developing are not primarily intended to describe the application to build. Instead, our models provide a high-level description of the information found in the computer during the project life cycle. A large fraction of these models represent application components, as well as other concepts. In a CADSE system, the software expert specifies models and metamodels for describing how model elements are linked with computer artefacts. A model element may be mapped to a few lines in a file, or to large structures such as projects. We believe that synchronizing models and artefacts is a prerequisite to any MDE projects.

The next topic that hampers MDE is evolution control. As a model element can be potentially mapped to large software artefacts that need to be versioned, the model element has to be versioned as well. This raises the issues of versioning model elements and software artefacts in a consistent manner.

The design of our system is based on few principles. First, the developer must be oblivious of the complexity involved in versioning and in evolution control. To that end, the developer works in a workspace in which the concept of version does not appear. Versioning is automatically executed following the evolution strategy that has been defined. Versioning and evolution strategies are established once and for all by software experts at the metamodel level.

The system explicitly targets reuse. To that end, the versioning granularity is rather low (i.e. a single model element), while version proliferation is avoided. This feature ensures that all possible combinations of elements are possible. Among other things, the evolution strategy renders explicit the kind of versions to be created with respect to the changes performed and with respect to the semantic of the links established among objects. The evolution characteristics established on links allow the software expert to express what does mean compatibility and in which way changes propagate.

During the execution of the evolution strategy, the system creates the versions as required, but also automatically updates and records compatibility information. Based on the recorded compatibility information, the system is able to determine if an assembly is consistent, as well as to suggest which changes would be required for making the assembly consistent, or to assist the user in rendering a workspace consistent.

Completeness can be checked, with respect to some relationship types. Therefore, the system is capable to build, or to assess, complete and consistent assemblies with minimal knowledge from the developers.

We believe that this system, which has been daily in production over the last 3 years, is a significant step towards the practical use of the MDE approach in software engineering.

CADSEg can be downloaded from `http://cadse.imag.fr/`

References

1. Emf project, `http://www.eclipse.org/modeling/emf/?project=emf`
2. Tarr, P., Hailpern, B.: Model-driven development: The good, the bad, and the ugly. IBM Systems Journal 45(3), 451 (2006)
3. Estublier, J., Vega, G.: Reconciling software configuration management and product data management. In: ESEC-FSE 2007: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 265–274. ACM, New York (2007)
4. Lalanda, P., Leveque, T., Estublier, J., Vega, G.: Domain specific engineering environments. In: APSEC (2008)
5. Helsen, S., Czarnecki, K.: Classification of model transformation approaches. In: OOPSLA (2003)
6. Navathe Raji Ahmed, S.B.: Version management of composite objects in cad databases. ACM, New York (1991)
7. Westfechtel, B., Conradi, R.: Towards a uniform version model for software configuration management. LNCS. Springer, Heidelberg (1997)
8. Rochkind, M.J.: The source code control system. IEEE Trans. on Software Engineering, 364–370 (1975)
9. Schmidt, D.C.: Model-driven engineering, p. 7 (2006)

An xADL Extension for Managing Dynamic Deployment in Distributed Service Oriented Architectures

Mohamed Nadhmi Miladi[1], Ikbel Krichen[1], Mohamed Jmaiel[1],
and Khalil Drira[2,3]

[1] University of Sfax, ReDCAD laboratory, ENIS, Box.W 1173, 3038, Sfax, Tunisia
MohamedNadhmi.Miladi@isimsf.rnu.tn,
[2] CNRS ; LAAS ; 7 avenue du colonel Roche, F-31077 Toulouse, France
[3] Université de Toulouse ; UPS, INSA, INP, ISAE ; LAAS ; F-31077 Toulouse, France

Abstract. In this paper, we present "3DxSoAdl" an xADL extension for Managing Dynamic Deployment in Distributed Service Oriented Architectures (SOA). This extension describes the deployment and the management process of SOA architectures thanks to three major parts. The first, describes, in the repository container, the services to be deployed. The second, describes the distribution of the services over the deployment nodes. The last part describes how to manage the dynamic evolving of architectures based on redeployment actions.

Keywords: SOA, SCA, xADL extension, Distributed deployment description, Deployment management.

1 Introduction

Software architectures today increasingly operate in large scale distributed systems with variable and unpredictable execution context as well as continuous user requirement discrepancy. In compliance with this trend, the deployment of these architectures should be managed to react with resource variability, user needs change and system faults.

Managing architecture deployment at development level is costly to build, difficult to modify, and usually provides only partial handling of system faults. Dynamically evolving deployment architectures should be managed following a set of well defined rules elaborated during the design process [1]. This requires using appropriate design techniques, models, languages and tools, to reason about the architecture's dynamic evolving. In this challenging research area, Service-Oriented Architecture (SOA) approach seems to be a promising technical approach. SOA enables dynamic evolution of applications, making responses for adaptiveness interaction to new possible needs [2].

In this paper, we make a strong case in favor of describing the deployment and its dynamic management of distributed service-oriented architectures. In this research direction, several works have been conducted in the field of Architecture Description Language (ADL). After a thorough study of these languages

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 439–446, 2010.

and their numerous proposed extensions, we adopted xADL which we extend to achieve the deployment description and the dynamic management of its evolving. In order to handle the deployment description of service-oriented architectures, we introduce additional notations to represent the deployment nodes, to describe the deployed services and to specify the repositories containing service descriptions. We also provide notations to describe the management actions needed to handle the dynamic management of the architecture deployment.

The proposed extension provides a flexible solution for specifying the ruling mechanisms for the SOA deployment and its management. The proposed notation also supports automatic refinement for generating platform-specific deployment actions for middleware implementing SOA technology, like OSGi. Our extension has been also integrated with higher notations approaches including many abstraction levels following the Model-Driven Architecture (MDA).

The rest of this paper is organized as follows: Section 2 presents our xADL extension. In section 3, we discuss some related works. Section 4 concludes this paper and presents future work directions.

2 The Deployment and Its Management Description in "3DxSOADL"

We present, in this section, an xADL extension called "3DxSoAdl" (Dynamic Distributed Deployment eXtensible Service Oriented Architecture Description Language). Compared to most proposed ADLs that focus on the configuration concept in a component-based architecture, our contribution adopts a modular approach based on a service-oriented paradigm supporting concepts such as registry, services . . . The description proposed, here, is more useful especially for an applicative point of view. It focuses on building and managing applications through a service-oriented approach. This vision follows various groups of vendors, including BEA, IBM, Oracle, SAP, and committee efforts including OASIS [3]. It is based on the "serviceComponent" notation which describes a software building block that exposes one or more services to potential requesters [4]. In addition, the proposed ADL extension highlights the deployment aspect of software architectures. The architecture deployment description provides a more suitable distributed vision of the architecture structure than a traditional configuration description. The deployment description stresses a transparent description of the deployment containers including node notation. Deployment containers description is crucial in a distributed context. It can be decisive in a further architecture implementation and in the choice of the architecture entities to deploy. Moreover, our contribution is based on standards including SCA and SOARM to cope with the wide diversity of the proposed notations by current ADLs. Finally, the described deployment structure will be dynamically managed to ensure a context change adaptation and to meet the various evolving requirements.

This xADL extension can be incorporated as a part of a refinement process, based on the MDA approach [5]. It layers various levels of abstraction to model and manage the SOA architecture deployment onto the underlying execution

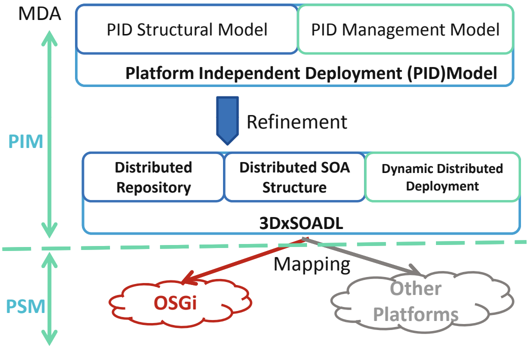


Fig. 1. The refinement process for deploying and managing software architectures

platform. The "3DxSoAdl" notations plays a mediator role between a highly abstract deployment and management models (Platform independent deployment model) and a platform-specific deployment and management description as depicted in Figure 1. Our description reduces the gap between an abstract modeling and its implementation. It provides (i) a modeling that traces a deployed architecture (ii) explicit redeployment actions that can be automatically translated into architectural management scripts. Tracing the deployed architecture and redeployment actions is extremely benefited to achieve statistics measurement, prevention actions, testing operations, and responsibility checking. In this paper, we focus only on describing the basic notations of the "3DxSoAdl" extension. Refinement aspects will be detailed in a more extended version.

This extension focuses on providing, on one hand, a description close to an SOA platform (such as OSGi, Jini,. . .) in order to model the SOA architecture deployment in a distributed context. On the other hand, it provides explicit description for dynamically managing the architecture deployment evolving.

Our contribution is composed of two major parts. The first, allows to describe the SOA architecture deployment and the second for describing the dynamic management of its evolving. In the first part, we provide deployment notations such as node notation. Moreover, we provide notations for describing SOA architecture deployment structure. This description is achieved using SOA standards such as "SOARM" [6] (Service Oriented Architecture Reference Model). More specifically, our extension is based on the SCA (Service Component Architecture) [4] specification that provides suitable solutions for SOA architectures description. In the second part, we describe the dynamic management of the architecture deployment evolving thanks to some management actions. This description is achieved through three XML schemas.

2.1 Distributed Repository Schema

This schema focuses on providing a description of all functional entities that can be useful for architecture deployment. It binds between them and their

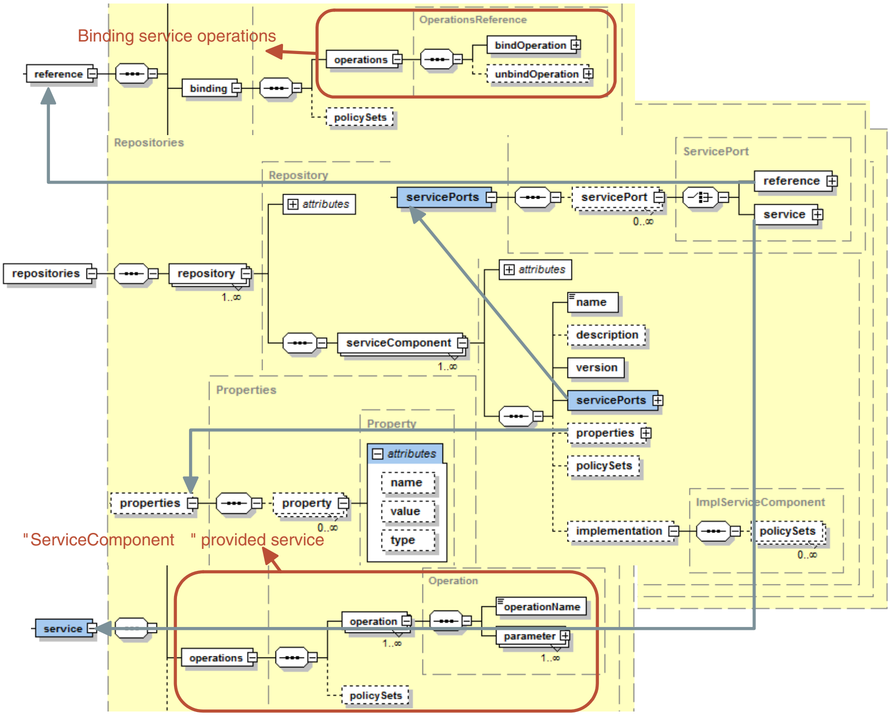


Fig. 2. Distributed repository schema

real implementation location. This binding is useful for the deployment and the management process. The achievement of this schema is basically inspired from SCA specification. The Distributed repository schema is composed of one or some distributed repositories. Each one of them belongs to a specific node and it includes the description of a set of “serviceComponents” as depicted in Figure 2. All schemas presented in this section are generated using the “XMLSpy” tool.

The “serviceComponent” description is crucial in various SOA tasks including the service discovery and service invocation. It is defined with two parts. The first, describes the functional aspect of “serviceComponent”. It includes a unique identifier called “scid”, a “serviceComponent” name, a set of “servicePorts” that describes “serviceComponent” interfaces (inspired from SOADL [7]) and a version number. In addition, we describe “serviceComponent” implementation in an abstract manner. It can be used to model some future implementation. This description extends xADL notations and more especially by its “implementation schema”. More specifically, a “serviceComponent” is a modular unit that handles a set of services. Their descriptions are involved in “servicePort” notation.

The “servicePort” can be either “service” or “reference” as depicted in Figure 2. First, “service” that represents “serviceComponent” provided services, has a service name and a set of operations containing called parameters. Second, “reference” that represents “serviceComponent” required services, has also a reference name. For each reference, we define two operations: bind and unbind. They enable the communication establishment and release between two “serviceComponents” through their “servicePorts”. These operations describe the basic operations to connect/disconnect a reference to one or many services. The cardinality binding is described with the multiplicity notation.

The second part focuses on non-functional aspects. A “serviceComponent” can also have a description that depicts its basic functionality and some “policySets” (referring to the SOA reference model standard). “PolicySets” describes the quality of services (QoS) and some constraints that can be applied in SOA architecture (such as security policies, transaction, reliable transmission of messages, encryption of messages, and so on). They can be divided into two types. The ”Interaction policies” that establish the contract between the providers and the consumers of the services (for example, the format of ”wires”, authenticity, confidentiality, and so on). The “Implementation policies”, which affect the contract between a component and its container (i.e. the manner in which the container must manage environmental component such as access control, monitoring). Moreover, it can own some properties.

2.2 Distributed SOA Structure Schema

This schema, called “archOSStructure”, ensures a double function: it models (i) the registry concept which is specific to the SOA approach. For each node of the application and for each service deployed in a given node, two references are established. The first references a description of the deployed service established in the previous schema. The second, references the invocation of the deployed service. This latter ensures SOA tasks including the publication and service discovery during the deployment phase. (ii) It describes the deployed distributed architecture structure. The “archOSStructure” schema, provides a deployment description from which we can either acquire a wide view of the overall system architecture deployment despite its complexity and its large-scale deployment. Indeed, this deployment view describes which “serviceComponents” are deployed, on which nodes they are deployed and which connections are established. This is depicted in Figure 3.

This schema is defined through a set of nodes owning some “serviceComponents” and connections within these “serviceComponents” called attachment, as depicted in Figure 3. First, “nodeRegistry” notation models active registry container. Each node is identified with “nodeId” and owns a set of deployed “serviceComponents”. For each “serviceComponent”, we provide a reference to its description. This description is useful in the service discovery task. We provide also “REF” which references the deployed service model ensuring its invocation after its discovery.

Second, attachment notation describes connections within the deployed “serviceComponents”. They can be either “attached” or “remoteAttached”. The first kind describes connections between “serviceComponents” within the same node (local attachment). It is described by the service and reference identification (called “idProv” and “idReq”) and the name of the “bindOperation” (defined in

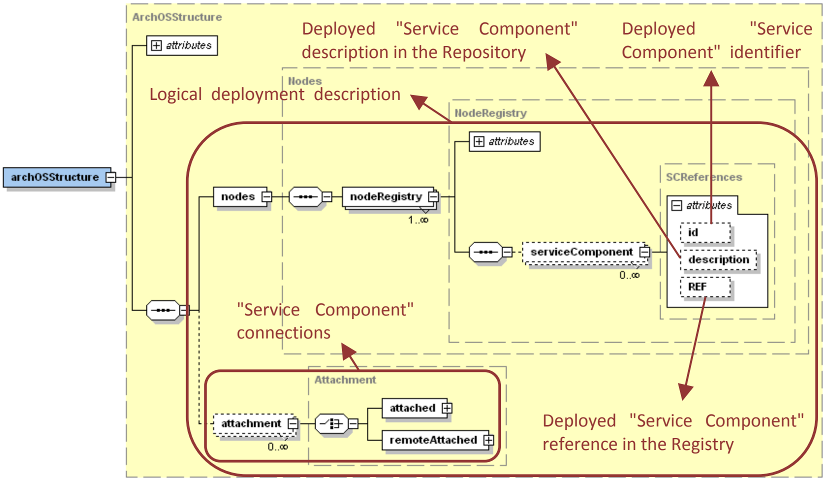

ArchOSStructure
attributes
Deployed "Service Component" description in the Repository
Deployed "Service Component" identifier
Nodes
Logical deployment description
NodeRegistry
archOSStructure
attributes
SCReferences
nodes
nodeRegistry
1..∞
attributes
id
serviceComponent
description
0..∞
REF
"Service Component" connections
Attachment
attached
attachment
0..∞
remoteAttached
Deployed "Service Component" reference in the Registry

Fig. 3. Distributed SOA structure schema

the reference notation) called “bindOperationName”. The second kind describes the connection between “serviceComponents” in different nodes. This latter extends the attached notation described earlier by modeling the provider node and the requested node.

2.3 Dynamic Distributed Deployment Schema

This schema, called “archOSConfiguration”, is designed to dynamically manage the architecture deployment evolving. This management is achieved through a set of management actions presented in the “archOSStructure” schema. These actions focus mainly on the structural management of the deployed architecture to meet the non-functional requirements.

These actions can be classified into two types: actions that affect the deployed architecture (“serviceComponents” deployment and connection) and actions that affect deployed structures (nodes and repositories). The first type involves “serviceComponent” deployment management actions. In fact, it includes “deploy” action that is described by “sourceLocation” representing the deployed “serviceComponent” name, “repositoryURL” representing the repository address and “nodeID” representing the node on which “serviceComponent” will be deployed. Moreover, it involves also “serviceComponents” connection management actions that own, on one hand, “attach” and “remoteAttach” (inherited from “archOSStructure” schema). In the other hand, “detach” and “remoteDetach” in their turn express “serviceComponent” disconnection. The second type, changes the architectural deployment structure. In fact, the architecture deployment nodes can be altered through the “addNode” and “deleteNode” actions. Moreover, the repository system can be changed using the “addRepository” and “deleteRepository” actions. In addition, “serviceComponent” can be either shifted within repositories system (depending on the system requirements) or modified by the “addServiceComponent” and “deleteServiceComponent” actions.

3 Related Works

Despite efforts in developing new Architecture Description Languages or proposing new extensions, most the achieved research efforts focus on component-based architectures with few solutions for SOA. SO-ADL [7] handles service oriented architectures. However, it stresses the behavior interaction description. Our extension completes this effort and provides means for the description of the SOA architecture deployment and its management. The challenge in describing architecture deployments is to provide solutions for dynamically manage this deployment. Research in this field can be sub-categorized in various classes. Works including [8] switch within only predefined configuration to dynamically manage their architecture. Other classes try to overcome this weakness by providing ADL extension including dynamic Acme [9] and Olan ADL [10]. But, some of these extensions still have some weakness in the dynamic description handling (OLAN does not permit the communication between the new instances and the primitive ones and dynamic Acme can not describe for instance the removal of components or connections). Some other classes are based either on new ADLs for dynamic aspect handling including C&C-ADL [11], or on the elaboration of ADL projects development including “Plastik” project [12] or “ArchWare” [13]. Despite all these research efforts, dynamic properties handled in the literature focus on the configuration aspect. The extension, proposed here, raises these efforts by handling the dynamic management deployment aspect including changing the architecture deployment within nodes.

4 Conclusion

In this paper, we presented an xADL extension named “3DxSoAdl” for describing the deployment and management activities for service oriented architectures. Using our extension, we can provide (i) a description that models a structural logical deployment in SOA architectures. This description is based on the specification standard SOARM and SCA (ii) it ensures requirement properties (i.e. QoS preservation), the context adaptation, the system failure recovery by dynamically managing the architecture deployment evolving. This management is achieved through the modeling of management actions that can react with the system context variations; (iii) it provides also a conceptual friendly description and flexible description notation. The “3DxSoAdl” extension involves notations that can easily be extended to various SOA implementations.

This extension provides notations to model the deployment and its management for SOA architecture independently from all specific technical implementations. To ensure the cohesion between the platform-independent models and the platform-specific models, a “3DxSoAdl” refinement has been developed. It translates SOA specifications described using “3DxSoAdl” notations into OSGi specifications and implements the management of the architecture deployment on specific OSGi platform such as Oscar, Felix or Equinox.

References

1. Meservy, T., Fenstermacher, K.: Transforming software development: an MDA road map. Computer 38(9), 52–58 (2005)
2. Carey, M.: Soa what? Computer 41(3), 92–94 (2008)
3. SCA: Service component architecture-assembly model specification-sca version 1.00 (2007), http://www.osoa.org/download/attachments/35/SCA/AssemblyModel/V100.pdf?version=1
4. Curbera, F.: Component contracts in service-oriented architectures. Computer 40(11), 74–80 (2007)
5. Miller, J., Mukerji, J.: MDA guide version 1.0.1. Technical report, Object Management Group OMG (2003)
6. SOARM: Service oriented architecture reference model, http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf
7. Jia, X., Ying, S., Zhang, T., Cao, H., Xie, D.: A new architecture description language for service-oriented architec. In: GCC 2007: Proceedings of the Sixth International Conference on Grid and Cooperative Computing, pp. 96–103. IEEE Computer Society, Los Alamitos (2007)
8. Allen, R., Vestal, S., Cornhill, D., Lewis, B.: Using an architecture description language for quantitative analysis of real-time systems. In: WOSP 2002: Proceedings of the 3rd international workshop on Software and performance, pp. 203–210. ACM, New York (2002)
9. Wile, D.: Using dynamic acme. In: Proceedings of a Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia (December 2001)
10. Bellissard, L., Ben Atallah, S., Boyer, F., Riveill, M.: Distributed application configuration. In: ICDCS 1996: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS 1996), Washington, DC, USA, pp. 579–585. IEEE Computer Society, Los Alamitos (1996)
11. Cîmpan, S., Leymonerie, F., Oquendo, F.: Handling dynamic behaviour in software architectures. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, pp. 77–93. Springer, Heidelberg (2005)
12. Batista, T.V., Joolia, A., Coulson, G.: Managing dynamic reconfiguration in component-based systems. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, pp. 1–17. Springer, Heidelberg (2005)
13. Oquendo, F., Warboys, B., Morrison, R., Dindeleux, R., Gallo, F., Garavel, H., Occhipinti, C.: Archware: Architecting evolvable software. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, pp. 257–271. Springer, Heidelberg (2004)

A First Step towards Security Policy Compliance of Connectors

Sun Meng

CWI, Kruislaan 413, Amsterdam, The Netherlands
M.Sun@cwi.nl

Abstract. Connectors have emerged as a powerful concept for composition and coordination of concurrent activities encapsulated as components and services. The widespread use of connectors in service-oriented applications is hindered by the lack of adequate security support. A security policy defines a set of security requirements that correspond to permissions, prohibitions and obligations to some executions when some contextual conditions are satisfied. In this paper, we propose the use of a scenario-based visual notation, called *Policy Sequence Chart* (PSC), for specifying security policies, and investigate an approach in which connectors are compliant with the security policies.

Keywords: Security Policy, Policy Sequence Chart, Connector, Compliance, Reo.

1 Introduction

Service-oriented computing (SOC) [12] is an emerging paradigm for the development of complex applications that may run on large-scale distributed systems. Such systems, which typically are heterogeneous and geographically distributed, usually exploit communication infrastructures whose topology frequently varies and components can, at any moment, connect to or detach from. Compositional coordination models and languages provide a formalisation of the "glue code" that interconnects the constituent components/services and organises the communication and cooperation among them in a distributed environment. They support large-scale distributed applications by allowing construction of complex component connectors out of simpler ones. An example of such a model is Reo [2], which offers a powerful glue language for implementation of coordinating component connectors based on a calculus of channels.

As organizations increase their use of services and adopt them as the primary building blocks to construct fairly complex distributed applications, security policy disclosure become crucial. The widespread adoption of coordination mechanisms in service-oriented applications can not happen without proper solutions for security problems. In this paper, we propose to use a scenario-based visual notation, called *Policy Sequence Chart* (PSC) to specify security policies in service coordination. A security policy is specified as a set of PSCs which describes the behavior that should be guaranteed / prohibited by a connector.

Once a security policy of a connector is specified, it should be enforced in the target connector implementation. This can be achieved by deploying the relevant security

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 447–454, 2010.

mechanisms and monitoring the execution steps of the target connector. Then the compliance of the connector with the policy should be checked.

Many works in the last years investigate the problem of modeling policies and compliance rules. For example, Schneider [16] uses security automata for specifying enforceable security policies and discusses mechanisms for enforcing security policies specified by such automata, which can be applied to safety-critical systems. Access control lists (ACLs) are used in [15] to capture simple policies in an unambiguous way. However, ACLs lack the expressive power being needed by providers. Another option is to write policies in an XML-based language, such as ODRL [14], which can be used for securely specifying and managing rights and conditions. Unfortunately, most of such languages lack formal semantics and make policies written in them ambiguous. Governatori et al. [8] developed a Formal Contract Language (FCL) for representing compliance requirements extracted from service contracts. FCL expresses normative behavior of the contract signing parties by means of chains of permissions, obligations, and violations. Brunel et al. [6] use Labeled Kripke Structures (LKS) which are a state/event extension of LTL both for specifying system behavior and related security requirements, also defined in a form of permissions, obligations, and violations. Recently, visual specifications of security policies written in modeling notations such as UML and graph transformation are studied. For example, Koch et al. use UML object diagrams and OCL constraints for specifying a role-based access control policy in [11].

In this paper, we propose using a scenario-based visual notation, termed *Policy Sequence Chart* (PSC), for specifying security policies, and investigate an approach in which connectors are compliant with the security policies. Within the PSC language, a policy is seen as a set of message exchage sequences. The language can be used to describe permissions, prohibitions, obligations and violations. We allow the users to explicitly specify forbidden scenarios, which can only be implicitly defined in LSCs. Furthermore, the mandatory behavior in PSCs is separated into weak and strong obligations, where weakly obligated behavior can be violated and the system can still be considered as compliant with the policy if the corresponding sanctions are specified and enforced. The semantics for PSCs can be defined by constraint automata and be used to check the compliance between connectors and security policies that should be enforced by the connectors.

The remainder of this paper is organized as follows. Section 2 presents the basic elements in the PSCs language. In Section 3 we discuss the problem of compliance of Reo connectors with respect to the security policies specified by PSCs. Finally, Section 4 concludes the paper.

2 Policy Sequence Charts

2.1 Sequence Charts

In recent years, scenario based languages such as UML Sequence Diagrams (SDs) [13], message sequence charts (MSCs) [9,10], and LSCs [7], are being widely used to capture behavioral requirements of applications. Sequence charts are the core of these widely accepted notations, which represent a global view of interactions among the components (in the broadest sense) inside a system. Each sequence chart corresponds to a

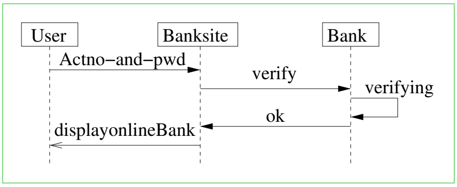


Fig. 1. A Permitted Policy Sequence Chart (pPSC)

single temporal sequence of interactions among system components/services and provides a partial system description. Sequence charts are close to users' understanding and they are often employed to refine use cases and to provide an abstract view of the system behavior. A sequence chart has two dimensions: a horizontal dimension representing the components participating in the scenario, and a vertical one representing time, i.e., the component temporal evolution or *lifeline*, represented by a vertical line. Actually the focus of a sequence chart lies on interactions between components by means of arrows. Such interactions, referred to as messages between a number of lifelines during a system run, define particular communications between lifelines of an interaction and can represent synchronous or asynchronous communications between components.

Figure 1 describes the basic syntax of a sequence chart. It captures a scenario in which a customer uses an online banking service of a bank and log in to the system. The customer keys in the account number and the password and the bank site sends customer information to the bank. Then the bank verifies the correctness of the information. Once the bank clears the customer, the banksite displays the online banking webpage to the customer.

2.2 Policy Sequence Charts

The main goal of PSC is to develop a scenario-based visual language to specify security policies of connectors and to allow compliance analysis. A security policy might concern permissions / prohibitions / obligations of a system behavior ("*it is permitted / forbidden / obligatory that...*"). For example, in an online bank system, we can have the following policies:

- A customer is permitted to transfer to another account only if the amount of the transfer is not more than 5000 euros.
- The system is obliged to stop the internet transaction of a bank customer if the user has been idle for more than 5 minutes.

In more complex situations, some security requirements can be violated while the system may still be considered compliant with the policy. For example, in the online bank system, a security rule specifies that a customer is forbidden to perform an account transfer if the account balance is negative after the transfer. According to the business model, there may be different possible implementations of this security rule. One compliant implementation would be, for some customers, to block any attempt to transfer an amount more than the account balance. However, for some other customers, another implementation would be to accept violations of the rule and implement a sanction, such as paying bank charges when the customer's account balance is negative. The second implementation should also be compliant with the policy if the policy specifies that the customer is obliged to pay bank charges when the account balance is negative.

Due to the different types of security policies, there are 4 types of PSCs: *permitted policy sequence charts* (pPSCs), *forbidden policy sequence charts* (fPSCs), *obligatory policy sequence charts* (oPSCs) and *sanction for violating policy sequence charts* (sPSCs). Each PSC consists of two parts: a main chart Ξ and a prechart p (or a guard condition p). The main chart is activated by the prechart or when the guard condition is satisfied. Sometimes the activating condition is weak, possibly degenerating to **true**, and the prechart itself will be even empty. For such cases, we may omit the second part and only use the main chart to specify the PSC.

Figure 1 shows an example of a pPSC, which is a basic sequence chart depicted in a (green) frame. The chart describes a possible behavior of the system and needs not be satisfied in all system runs. We only require that at least one run satisfies it.

Figure 2 and 3 are two examples of fPSCs. In Figure 2 we use a prechart to specify a sequence of events. Once the prechart is traversed successfully, the behavior described by the main chart is prohibited and can not be satisfied by any system run. Another option for the trigger of the forbidden behavior is to use a guard condition, as shown by Figure 3. The main chart for fPSCs, depicted by dashed (red) borderlines, can be of type *strong prohibition* or *weak prohibition*, denoted by the keywords **strong** or **weak** labeling the frame for the main chart respectively. The behavior specified by a strong prohibition can not happen, while weak prohibition can be violated, if corresponding sanctions are specified by some sPSCs, and the sanctions are enforced.

Figure 4 describes an example of an oPSC. In the chart we use a prechart to describe the sequence of events activating the obligated behavior. The prechart is shown in the upper part of the figure (in dashed line style). Thus, whenever the message sequence in

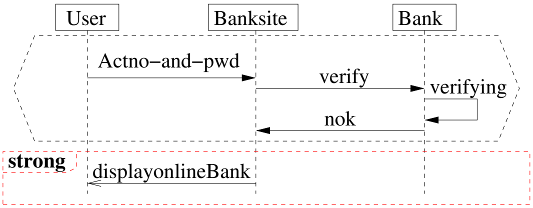


Fig. 2. A Forbidden Policy Sequence Chart (fPSC) with Prechart

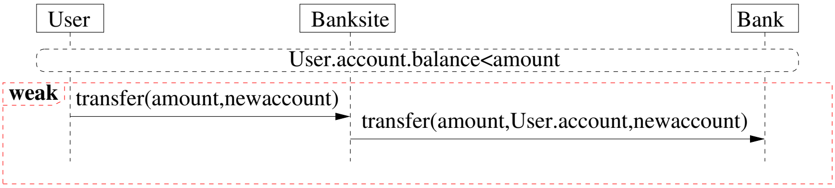

User
Banksite
Bank
User.account.balance<amount
weak
transfer(amount,newaccount)
transfer(amount,User.account,newaccount)

Fig. 3. A fPSC with Guard Condition

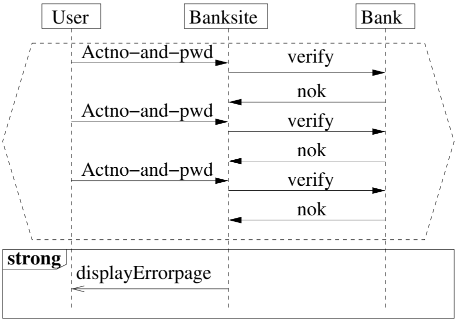


Fig. 4. An Obligatory Policy Sequence Chart (oPSC) with Prechart

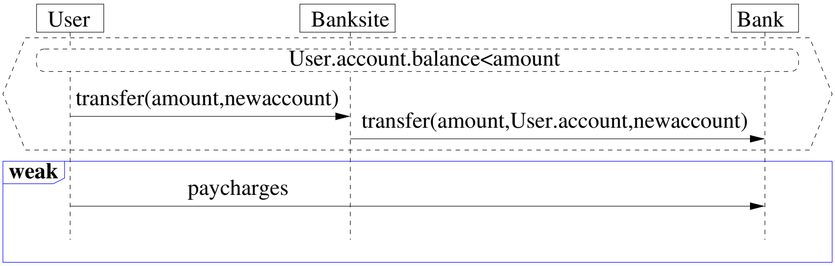

User
Banksite
Bank
User.account.balance<amount
transfer(amount,newaccount)
transfer(amount,User.account,newaccount)
weak
paycharges

Fig. 5. A Sanction for Violating Policy Sequence Chart (sPSC)

the prechart occurs, i.e., the customer fails 3 times to login the online bank system, the banksite should display an error page to the customer. Similar like fPSCs, the main chart of an oPSC can be of type *strong obligation* or *weak obligation*, denoted by the keywords **strong** or **weak** labeling the frame for the main chart respectively. Security policies specified with strong oPSCs can not be violated. Policies specified with weak oPSCs can be violated by a system which is still compliant with the policies, if corresponding sanctions are specified with some sPSCs, and the sanctions are enforced by the system.

A sPSC specifies the sanction when some violation happens. Actually, two types of violations may occur: a forbidden behavior that happens, or a (weakly) obligatory behavior that does not happen. The prechart of the sPSC specifies the violation, which can be the sequence chart in a given fPSC, or a confliction with a given oPSC. For example, the sPSC given in Figure 5 specifies the sanction when the prohibited behavior in Figure 3 happens. The sanction is captured by the main chart, which is depicted in a (blue) frame and can still be of type *strong obligation* or *weak obligation*, denoted by the keywords **strong** or **weak**. Sanctions specified as a weak obligation can also be violated.

3 Compliance of a Connector with Its Security Policy

Here we consider connectors specified as Reo circuits. Reo [2] is a channel-based exogenous coordination model wherein *connectors* are compositionally constructed from

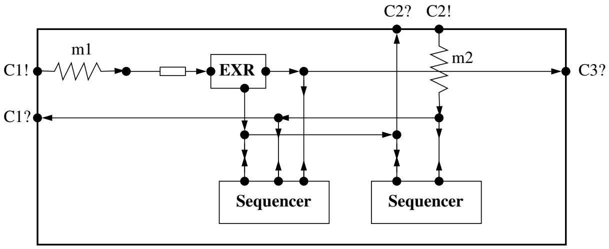


Fig. 6. An Example Connector among Three Components

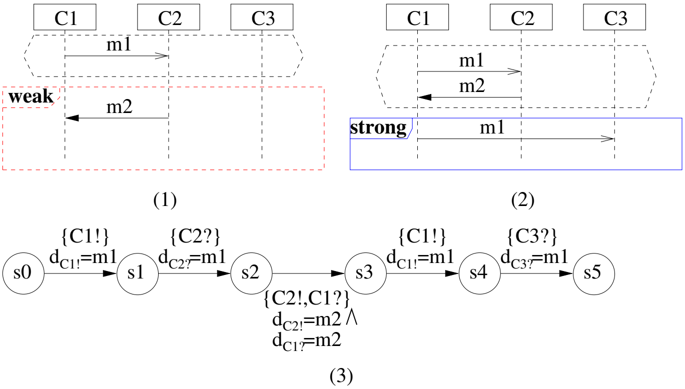


Fig. 7. Security Policy and Behavior of the Example Connector

simpler ones. Details about Reo and its semantics can be found in [2,3,4]. Complex connectors in Reo are organized in a network of primitive connectors, called *channels*. A connector provides the protocol that controls and organizes the communication, synchronization and cooperation among the components/services that they interconnect. Figure 6 is an example connector which interconnects three components. **EXR** and **Sequencer** used in Figure 6 are exclusive router and sequencer, respectively.

A security policy $\mathscr{P}$ is specified by giving a set of PSCs. Given a connector and a policy, we focus on the meaning of compliance of the connector with the policy in this section.

As an example, we present part of the connector among three components $C1$, $C2$ and $C3$ as in Figure 6. The communication among the components encoded in the connector is specified by the constraint automata in Figure 7 (3). The security policy consists of the two PSCs given in Figure 7 (1) and (2). The first PSC specifies the weakly forbidden behavior. If $C1$ sends an asynchronous message $m1$ to $C2$, it is prohibited to receive a synchronous message $m2$ from $C2$ later. The sequence of the first 3 transitions is performed and violates the fPSC in Figure 7 (1). So, according to (2), there is a strong obligation for $C1$ to send an asynchronous message $m1$ to $C3$ as a sanction. This is performed by the specified behavior in (3). Thus, according to our approach, the connector is compliant with its security policy, even if the first rule is violated.

We say that a non-forbidden policy (i.e., a policy specified by pPSC, oPSC or sPSC) is *fulfilled* by a connector if the event sequences given in the chart is going to be performed from the current state of the connector, and a forbidden policy specified by fPSC is *fulfilled* by a connector if the event sequences given in the chart is not performed from any state of the connector. For example, the obligation for $C1$ to send the asynchronous message $m1$ to $C3$ in Figure 7 (2) is fulfilled at state $s3$ in Figure 7 (3).

A violation may induce a sequence of further obligations. We use $Wsanc(\circledast, \eta, s_i)$ to denote the rule which is weakly obligatory in state s_i of a run r because of the violation of η, where $\circledast \in \{o, f\}$ denotes the type of the violation of η (obligations being not fulfilled or forbidden behavior occurred). A sequence of sanctions is a sequence of sPSCs $\langle P_i = (p_i, \Xi_i)\rangle_{i\in I}$, where the last obligation (in case of finite sequence) type can be either weak or strong obligation, and all the other obligations are weak. A sequence $\langle P_i\rangle_{i\in I}$ of sanctions is triggered by violation of η if the first sanction in the sequence is $Wsanc(\circledast, \eta, s_i)$ for violation of a sequence η at some state s_i, and every of the successive sanction in the sequence is triggered by the violation of the previous sanction. If a sequence of sanctions triggered by a violation ends with a fulfilled obligation, then the violation is called *managed*. A connector is compliant with a policy if

- either there is no violation,
- or each time some violation happens, the associated sanction is enforced. In other words, every sequence of violations is managed.

4 Conclusion and Future Work

In this paper we propose a formalism for specifying and managing security policies that apply when some security requirements are violated, which is called *Policy Sequence Chart* (PSC), and investigate the compliance of connectors with security policies specified as PSCs. Within PSC a security policy is specified as a set of PSCs, including scenarios for permitted, prohibited and obligated communications. Violation of policies has been investigated in [6] by using Labeled Kripke Structures, but the approach does not consider explicit prohibitions and the violations only apply to single transitions. Comparing to other approaches, the PSC language is simple, but sufficiently expressive and user-friendly for capturing security requirements on connectors.

As future work we plan to introduce timing constraints so that we can deal with obligations with deadline as discussed in [5]. The notion of compliance between connectors and policies should also be further investigated to consider more complex requirements. We also intend to conduct some larger case studies to validate the approach. The existing tools, for example, the Reo model checker [1], may be useful to check for compliance of connectors with respect to security policies.

Acknowledgements. The work reported in this paper is supported by a grant from the GLANCE funding program of the Dutch National Organization for Scientific Research (NWO), through project CooPer (600.643.000.05N12).

References

1. Eclipse Coordination Tools, `http://reo.project.cwi.nl/`
2. Arbab, F.: Reo: A Channel-based Coordination Model for Component Composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
3. Arbab, F., Rutten, J.: A coinductive calculus of component connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2003. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003)
4. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. Science of Computer Programming 61, 75–113 (2006)
5. Brunel, J., Bodeveix, J.-P., Filali, M.: A state/event temporal deontic logic. In: Goble, L., Meyer, J.-J.C. (eds.) DEON 2006. LNCS (LNAI), vol. 4048, pp. 85–100. Springer, Heidelberg (2006)
6. Brunel, J., Cuppens, F., Cuppens-Boulahia, N., Sans, T., Bodeveix, J.-P.: Security policy compliance with violation management. In: Proc. of the Workshop on Formal Methods in Security Engineering (FMSE 2007), pp. 31–40. ACM Press, New York (2007)
7. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Formal Methods in System Design 19(0) (2001)
8. Governatori, G., Milosevic, Z., Sadiq, S.: Compliance checking between business processes and business contracts. In: Proceedings of the International Enterprize Distributed Object Computing Conference (EDOC 2006), pp. 221–232. IEEE Computer Society, Los Alamitos (2006)
9. ITU-TS. Recommendation Z.120 : Message Sequence Chart (MSC), Geneva (1996)
10. ITU-TS. Recommendation Z.120(11/99) : MSC 2000, Geneva (1999)
11. Koch, M., Parisi-Presicce, F.: Visual Specifications of Policies and Their Verification. In: Pezzé, M. (ed.) FASE 2003. LNCS, vol. 2621, pp. 278–293. Springer, Heidelberg (2003)
12. Papazoglou, M.P., Georgakopoulos, D.: Service Oriented Computing. Comm. ACM 46(10), 25–28 (2003)
13. Object Management Group. Unified Modeling Language: Superstructure - version 2.1.1 (2007), `http://www.uml.org/`
14. ODRL: The Open Digital Rights Language Initiative, `http://odrl.net`
15. Pfleeger, C.P.: Security in Computing. Prentice Hall, New Jersey (1997)
16. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security 3(1), 30–50 (2000)

A Safe Implementation of Dynamic Overloading in Java-Like Languages*

Lorenzo Bettini[1], Sara Capecchi[1], and Betti Venneri[2]

[1] Dipartimento di Informatica, Università di Torino
{bettini,capecchi}@di.unito.it
[2] Dipartimento di Sistemi e Informatica, Università di Firenze
venneri@dsi.unifi.it

Abstract. We present a general technique for extending Java-like languages with dynamic overloading, where method selection depends on the dynamic type of the parameter, instead of just the receiver. To this aim we use a core Java-language enriched with encapsulated multi-methods and dynamic overloading. Then we define an algorithm which translates programs to standard Java code using only basic mechanisms of static overloading and dynamic binding. The translated programs are semantically equivalent to the original versions and preserve type safety.

Keywords: Language extensions, Multi-methods, Dynamic Overloading.

1 Introduction

A *multi-method* can be seen as a collection of overloaded methods, called *branches*, associated to the same message, but the method selection takes place dynamically according to the run-time types of both the receiver and the arguments, thus implementing *dynamic overloading*. Though multi-methods are widely studied in the literature, they have not been added to mainstream programming languages such as Java, C++ and C#, where overloading resolution is a static mechanism (the most appropriate implementation of an overloaded method is selected statically by the compiler according to the static type of the arguments).

In this paper we present a general technique for extending Java-like languages with dynamic overloading. To this aim we use the core languages FDJ and FSJ which are extensions of Featherweight Java [7] with multi-methods and static overloading, respectively. Then, we define an algorithm which translates FDJ programs to FSJ code using only basic mechanisms of static overloading and dynamic binding. Both FDJ and FSJ are based on [2], where we studied an extension of FJ with static and dynamic overloading from the linguistic point of view; thus, in that paper, we focused on the type system and the crucial conditions to avoid statically any possible ambiguities at run-time. The multi-methods we are considering are *encapsulated* in classes and not external functions, and the branch selection is *symmetric*: during dynamic overloading

* This work has been partially supported by the MIUR project EOS DUE.

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 455–462, 2010.

selection the receiver type of the method invocation has no precedence over the argument types.

The translation here presented can be regarded as a formalization of a general technique to implement dynamic overloading in mainstream object-oriented languages. The translation is type safe (i.e., the generated code will not raise type errors) and the translated code will have the same semantics of the original program using dynamic overloading. In particular, since the translated code uses only static overloading and dynamic binding, it does not introduce a big overhead (it performs method selection in constant time, independently from the width and depth of inheritance hierarchies) as in other approaches in the literature [4,6]. In [3] we presented `doublecpp`, `http://doublecpp.sf.net`, a preprocessor for C++ which is based on the approach here presented ([3] also sketches a very primordial and informal version of the translation algorithm); the translation presented in this paper is the first formalization of our technique for implementing dynamic overloading through static overloading.

We briefly present the parts of the core language FDJ (*Featherweight Java + Dynamic overloading*), which is an extension of FJ (*Featherweight Java*) [7] with multi-methods, that are relevant for the translation algorithm (we refer to [2] for further details). In the following we assume that the reader is familiar with FJ, then we will concentrate on the features of FDJ. The syntax of FDJ is the following:

$$\begin{array}{llr}
\texttt{L} ::= & \texttt{class C extends C } \{\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K; }\overline{\texttt{M}}\} & \text{classes} \\
\texttt{K} ::= & \texttt{C(}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{)}\{\texttt{super(}\overline{\texttt{f}}\texttt{); this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;}\} & \text{constructors} \\
\texttt{M} ::= & \texttt{C m (C x)}\{\texttt{return e;}\} & \text{methods} \\
\texttt{e} ::= & \texttt{x} \;\big|\; \texttt{e.f} \;\big|\; \texttt{e.m(e)} \;\big|\; \texttt{new C(}\overline{\texttt{e}}\texttt{)} & \text{expressions} \\
\texttt{v} ::= & \texttt{new C(}\overline{\texttt{v}}\texttt{)} & \text{values}
\end{array}$$

The distinguishing feature, w.r.t. FJ, consists in the definition of multi-methods: the programmer is allowed to specify more than one method with the same name and different signatures; any definition of a method `m` in a class `C` is interpreted as the definition of a new branch of the multi-method `m` (this difference w.r.t. FJ is evident in the typing [2]). The new branch is added to all the other branches of `m` that are inherited (if they are not redefined) from the superclasses of `C` (*copy semantics of inheritance* [1]). In FDJ we also permit method redefinition with a covariant return type, a feature that has been recently added to Java. In the present paper we limit multi-methods to one parameter. We do not see this as a strong limitation from a pragmatic point of view. Indeed, most of the examples found in the literature dealing with multi-methods consider only one parameter.

A program is a pair $(CT, \texttt{e})$ of a class table (mapping from class names to class declarations) and an expression `e` (the program's main entry point). The subtyping relation $<:$ on classes (types) is induced by the standard subclass relation. The types of FDJ are the types of FJ extended with *multi-types*, representing types of multi-methods. A multi-type is a set of arrow types associated to the branches of a multi-method, and is of the shape: $\{\texttt{C}_1 \rightarrow \texttt{C}'_1, \ldots, \texttt{C}_n \rightarrow \texttt{C}'_n\}$. We will write multi-types in a compact form, by using the sequence notation: $\{\overline{\texttt{C} \rightarrow \texttt{C}'}\}$. Σ will range over multi-types. We extend the sequence notation also to multi-method definitions: $\overline{\texttt{C}'\ \texttt{m}\ (\texttt{C x})\{\texttt{return e;}\}}$ represents a sequence of method definitions, each with the same name `m` but with different signatures (and possibly different bodies): $\mathtt{C}'_1\ \mathtt{m}\ (\mathtt{C}_1\ \mathtt{x})\{\mathtt{return}\ \mathtt{e}_1;\}\ \ldots\ \mathtt{C}'_n\ \mathtt{m}\ (\mathtt{C}_n\ \mathtt{x})\{\mathtt{return}\ \mathtt{e}_n;\}$. The multi-type of the above multi-method will be denoted by $\{\overline{\mathtt{C} \rightarrow \mathtt{C}'}\}$.

Multi-types are constrained by two crucial consistency conditions, formulated in [5], which must be checked statically, in order to be well-formed: a multi-type $\{\overline{\mathtt{B} \rightarrow \mathtt{B}'}\}$ is *well-formed* if $\forall\ (\mathtt{B}_i \rightarrow \mathtt{B}'_i), (\mathtt{B}_j \rightarrow \mathtt{B}'_j) \in \{\overline{\mathtt{B} \rightarrow \mathtt{B}'}\}$ the following conditions are verified: 1) $\mathtt{B}_i \neq \mathtt{B}_j$, 2) $\mathtt{B}_i <: \mathtt{B}_j \Rightarrow \mathtt{B}'_i <: \mathtt{B}'_j$. The first condition requires that all input types are distinct. The second condition guarantees that a branch specialization is safe: if statically a branch selection has a return type, and if dynamically a more specialized branch is selected, the return type is consistent with the static selection (it is a subtype). The original definition of well-formedness of [5] also contained a third condition, which is always implied by the other two conditions in our context where we only have single inheritance and one parameter.

The $\mathit{mtype}(\mathtt{m}, \mathtt{C})$ lookup function returns the type of m in the class C which is a multi-type. In particular, since we consider copy semantics of inheritance, the multi-type contains both the signatures of the branches defined (or redefined) in the current class and the ones inherited by the superclass:

$$\frac{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\} \qquad \overline{\mathtt{B}'\ \mathtt{m}\ (\mathtt{B}\ \mathtt{x})\{\mathtt{return}\ \mathtt{e};\}} \in \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \{\overline{\mathtt{B} \rightarrow \mathtt{B}'}\} \cup \{\mathtt{B}_i \rightarrow \mathtt{B}'_i \in \mathit{mtype}(\mathtt{m}, \mathtt{D}) \mid \forall \mathtt{B}_j \rightarrow \mathtt{B}'_j \in \{\overline{\mathtt{B} \rightarrow \mathtt{B}'}\}, \mathtt{B}_i \neq \mathtt{B}_j\}}$$

$$\frac{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \mathit{mtype}(\mathtt{m}, \mathtt{D})}$$

Note that we cannot implement $\mathit{mtype}(\mathtt{m}, \mathtt{C})$ simply as $\{\overline{\mathtt{B} \rightarrow \mathtt{B}'}\} \cup \mathit{mtype}(\mathtt{m}, \mathtt{D})$ due to possible method overriding with covariant return types.

The semantics of method invocation is type driven in that it uses *mtype* to select the most specialized version among the set of matching branches (it is unique, by well-formedness, if that set is not empty). The selected method body is not only the most specialized w.r.t. the argument type but also the most redefined version associated to that signature. This way, we model standard method overriding inside our mechanism of dynamic overloading. The language FSJ has the same syntax as FDJ but the multi-methods are intended as standard overloaded methods and then the overloading resolution is static.

2 From FDJ to FSJ: The Translation Algorithm

We now use FDJ and FSJ to formalize the transformation from an extended Java with dynamic overloading to standard Java (with static overloading): in this section we show how *multi-methods* can be implemented by static overloading and dynamic binding. Our goal is to define a general technique to extend a language with dynamic overloading. The solution presented here is inspired by the one described by Ingall in [8] (on which also the Visitor pattern is based), but it does not suffer from possible implementation problems when implementing manually this technique.

We provide a translation algorithm that, given an FDJ program using dynamic overloading, produces an equivalent FSJ program only using static overloading and dynamic binding. This translation is thought to be automatically executed by a program translator (a preprocessor) that has to be run before the actual language compiler. Note that the code generated by our translation uses neither RTTI nor, more importantly, type downcasts which are very common in other proposals and that are notoriously sources of type safety violations. Thus, we provide a formal framework to reason about correctness of compilers when adding dynamic overloading to a language.

In order to give an informal idea of the proposed translation, let us consider the following example (for simplicity, in the following, we will use the full Java syntax, e.g., assignments and sequentialization).

Suppose we have the following FDJ program, where $\mathtt{B}_2 <: \mathtt{B}_1$ are not shown:

```
class A1{
     C m (B1 x){return e1;}          A1 z = new A1();
     C m (B2 x){return e2;}          B1 y = new B2();
};                                   z.m(y);
```

Then we consider the semantics of the method invocation `z.m(y)`: it will select the second branch of `m` in $\mathtt{A}_1$ since, in spite of being declared statically as $\mathtt{B}_1$, `y` is of type $\mathtt{B}_2$ dynamically.

Let classes $\mathtt{A}_1$, $\mathtt{B}_1$ and $\mathtt{B}_2$ be written in FSJ as follows:

```
                     class A1{
                       C m (B1 x){return x.disp_m(this);}
                       C m (B2 x){return x.disp_m(this);}
                       C _m (B1 x){return e1;}
                       C _m (B2 x){return e2;}
                     }
class B1{                                      class B2 extends B1{
  // original contents                           // original contents
  C disp_m (A1 x){return x._m(this);}            C disp_m (A1 x){return x._m(this);}
}                                              }
```

Summarizing, all method definitions are renamed by adding the (reserved) prefix _ and all the branches of the original multi-methods (including the ones implicitly inherited with copy semantics) are modified using the forward invocation `x.disp_m(this)`. Now let us analyze how the method invocation `z.m(y)` proceeds in FSJ:

1. `z.m(y)` will select the branch $\mathtt{C\ m\ (B_1\ x)\{return\ x.disp\_m(this);\}}$ in $\mathtt{A}_1$ (remember that `y` is statically of type $\mathtt{B}_1$);
2. `x.disp_m(this)` will select the (only) branch of `disp_m` in $\mathtt{B}_2$, since dynamic binding is employed also in the static overloading invocation;
3. `x._m(this)` in $\mathtt{B}_2$ will use static overloading, and thus will select a branch of `_m` in $\mathtt{A}_1$ according to the static type of the argument: the argument `this` is of type $\mathtt{B}_2$ and thus the second branch of `_m` in $\mathtt{A}_1$ will be selected.

Therefore, `z.m(y)` in the translated FSJ program, where classes are modified as above, has the same behavior as in the FDJ original program (since they execute the same method body $\mathtt{e}_2$). Consider the same classes of the previous example and the following additional classes (where $\mathtt{B}_3 <: \mathtt{B}_2$ and $\mathtt{C}' <: \mathtt{C}$):

```
class A2 extends A1{            A1 z = new A2();
   C' m (B3 x){return e3;}      B1 y = new B3();
};                              z.m(y);
```

The dynamic overloading semantics will select the branch $\mathtt{C}'$ $\mathtt{m(B_3\ x)}$ in $\mathtt{A_2}$. In this case the program would be translated in FSJ as follows (the translation of class $\mathtt{A_1}$ is the same as before):

```
class A2 extends A1{
   C m (B1 x){return x.disp_m(this);}
   C m (B2 x){return x.disp_m(this);}
   C' m (B3 x){return x.disp_m(this);}
   C' _m (B3 x){return e3;}
}
```

```
class B1{
   // original contents
   C disp_m (A1 x){return x._m(this);}
   C disp_m (A2 x){return x._m(this);}
}
```

```
class B2 extends B1{
   // original contents
   C disp_m (A1 x){return x._m(this);}
   C disp_m (A2 x){return x._m(this);}
}
```

```
class B3 extends B2{
   // original contents
   C' disp_m (A2 x){return x._m(this);}
}
```

Let us interpret the method invocation z.m(y) in FSJ:

1. As in the previous example, z.m(y) will select the branch C m ($\mathtt{B_1}$ x){return ...;} (remember that x is statically of type $\mathtt{B_1}$);
2. since dynamic binding is employed, the implementation of m in $\mathtt{A_2}$ will be selected dynamically;
3. the method invocation x.disp_m(this) will select statically the second branch of disp_m in $\mathtt{B_1}$, since this is (statically) of type $\mathtt{A_2}$, but since dynamic binding is employed, the version of such method provided in $\mathtt{B_3}$ will be actually invoked dynamically (note that disp_m in $\mathtt{B_3}$ is an override of disp_m in $\mathtt{B_2}$ with covariant return type, which is sound);
4. the method invocation x._m(this) in $\mathtt{B_3}$ will select a branch of _m in $\mathtt{A_2}$ according to the static type of the argument: the argument this is of type $\mathtt{B_3}$ and thus the branch $\mathtt{C}'$ ($\mathtt{B_3}$ x) of _m in $\mathtt{A_2}$ will be selected.

Again, z.m(y) in FSJ has the same behavior as in FDJ (they both execute the method body $\mathtt{e_3}$). The reader can easily verify that if y is assigned an instance of $\mathtt{B_2}$ we would execute $\mathtt{e_2}$, just as in the first example, i.e., the body of the branch with parameter $\mathtt{B_2}$ as defined in $\mathtt{A_1}$, implicitly inherited by $\mathtt{A_2}$. Summarizing, the idea of our translation is that the dynamic overloading semantics can be obtained, in a static overloading semantics language, by exploiting dynamic binding and static overloading twice: this way the dynamic selection of the right method is based on the run time types of both the receiver and the argument of the message.

Note that the key point in our translation is to rename by _m every method m and then introduce a new overloaded method m, which is the *entry point* for dynamic overloading interpretation. The branches of this new multi-method m in a FSJ class $\mathtt{A}_i$ are built starting from the branches of the original FDJ m by considering the set of all the parameters types $\mathtt{B}_j$ of m in $\mathtt{A}_i$ (including the ones inherited by copy semantics). One might be tempted to say that the added methods $\mathtt{C\ m\ (B_{1,2}\ x)\{return\ x.disp\_m(this);\}}$ in $\mathtt{A_2}$ are useless since they would be inherited from $\mathtt{A_1}$: however, their presence is fundamental to make our translated code work, since what matters in this context is the static type of `this`; thus, these methods must be present also in $\mathtt{A_2}$.

Our translation also relies on a new multi-method `disp_m` introduced in the $\mathtt{B}_i$'s classes. Let us consider how the branches of `disp_m` are built. Regarding the definition of the multi-method `m` in a FDJ class $\mathtt{A}_i$, for each type $\mathtt{B}_i$, such that $\mathtt{B}_i$ is the type of the parameter of a branch of `m`, we add a branch to `disp_m` in the class $\mathtt{B}_i$ with parameter of type $\mathtt{A}_i$. We add the same branch to `disp_m` in each class $\mathtt{B}_j$ such that $\mathtt{B}_i <: \mathtt{B}_j$ and $\mathtt{B}_j$ is the type of the parameter of a branch of `m` in $\mathtt{A}_i$ or in a superclass $\mathtt{A}_k$ ($\mathtt{A}_i <: \mathtt{A}_k$).

Let us consider the classes in the second example:

- $\mathtt{B_1}$ and $\mathtt{B_2}$ are the parameter types of the branches of `m` in $\mathtt{A_1}$, thus we add a branch to `disp_m` in $\mathtt{B_1}$ and $\mathtt{B_2}$, with parameter of type $\mathtt{A_1}$.
- $\mathtt{B_3}$ is the type of the parameter of the branch of `m` in $\mathtt{A_2}$ so we add a branch to `disp_m` in $\mathtt{B_3}$, with parameter of type $\mathtt{A_2}$. Moreover, since $\mathtt{B_3} <: \mathtt{B_2}$ and since $\mathtt{B_2}$ is the type of parameter of a branch of `m` in $\mathtt{A_1}$ ($\mathtt{A_2} <: \mathtt{A_1}$), we add a branch to `disp_m` in $\mathtt{B_2}$, with parameter of type $\mathtt{A_2}$; recursively, we add such a branch to $\mathtt{B_1}$, since $\mathtt{B_2} <: \mathtt{B_1}$ and since $\mathtt{B_1}$ is the type of parameter of a branch of `m` in $\mathtt{A_1}$ ($\mathtt{A_2} <: \mathtt{A_1}$).

To insert the correct `disp_` methods following the strategy above, it is enough to retrieve all the branches of a multi-method in a class (including those inherited with copy semantics); for each of such methods in a class $\mathtt{A}_i$, say $\mathtt{B'\ m(B}_j\ \mathtt{x)}\ \{\ldots\}$, we insert in $\mathtt{B}_j$ a method $\mathtt{B'\ disp\_m(A}_i\ \mathtt{x)\ \{return\ x.\_m(this);\}}$. It is easy to verify that this procedure adds exactly all and only the `disp_` that are required to make our translation work. Summarizing, the method `m`, in the FSJ translated version, aims at statically using the type of $\mathtt{A}_i$ and dynamically using the type of the $\mathtt{B}_j$, while the method `disp_m` has exactly the opposite task. Together these two methods realize the dynamic overloading semantics.

Let us now present formally our translation, which is defined on well-typed FDJ programs, so assuming properties concerning well typedness of programs. In the following we introduce some auxiliary notations: given a set of method definitions $\overline{\mathtt{M}}$, a method definition `M`, a method name `m`, a class table *CT* and $\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\,\overline{\mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\} \in$ *dom*(*CT*) we use the following notations:

- $\overline{\mathtt{M}} \setminus \mathtt{m} = \{\mathtt{B'\ m'(B\ x)\ \{return\ e;\}} \in \overline{\mathtt{M}} \mid \mathtt{m'} \neq \mathtt{m}\}$
- $\mathit{rename}(\overline{\mathtt{M}},\mathtt{m}) = (\overline{\mathtt{M}} \setminus \mathtt{m}) \cup \{\mathtt{B'\ \_m(B\ x)\ \{return\ e;\}} \mid \mathtt{B'\ m(B\ x)\ \{return\ e;\}} \in \overline{\mathtt{M}}\}$
- $\mathit{rename}(\mathit{CT},\mathtt{C},\mathtt{m})$ is the class table CT' that is obtained from *CT* by renaming the methods with name `m` in the class `C`, i.e., CT' is such that:
  - $\mathit{CT}'(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\,\overline{\mathtt{f}};\ \mathtt{K};\ \mathit{rename}(\overline{\mathtt{M}},\mathtt{m})\}$
  - $\forall \mathtt{C'} \in \mathit{dom}(\mathit{CT})$ such that $\mathtt{C'} \neq \mathtt{C}$, $\mathit{CT}'(\mathtt{C'}) = \mathit{CT}(\mathtt{C'})$
- $\mathit{CT}(\mathtt{C}) \hookleftarrow \mathtt{M}$ is the class table CT' obtained from *CT* by adding to `C` the method definition `M`, i.e., CT' is such that:
  - $\mathit{CT}'(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\,\overline{\mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\ \mathtt{M}\}$
  - $\forall \mathtt{C'} \in \mathit{dom}(\mathit{CT})$ such that $\mathtt{C'} \neq \mathtt{C}$, $\mathit{CT}'(\mathtt{C'}) = \mathit{CT}(\mathtt{C'})$

Definition 1 (Translation algorithm from FDJ to FSJ). *Let $p = (\mathit{CT},\mathtt{e})$ be well typed* FDJ *program, then the corresponding* FSJ *translated version, denoted by $(\overline{\mathit{CT}},\mathtt{e})$, is obtained from p by performing the following algorithm:*

```
1. $\overline{CT} := CT$
2. $\forall\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\} \in dom(CT)$
   (a) $\forall\mathtt{m} \in Names(\overline{\mathtt{M}})$
         i. $\overline{CT} := rename(\overline{CT}, \mathtt{C}, \mathtt{m})$
        ii. $\forall\mathtt{B} \longrightarrow \mathtt{B}' \in mtype(\mathtt{m}, \mathtt{C})$, using $CT$
            A. $\overline{CT} := \overline{CT}(\mathtt{B}) \twoheadleftarrow \mathtt{B}'\ \mathtt{disp\_m(C\ x)}\ \{\mathtt{return\ x.\_m(this);}\}$
            B. $\overline{CT} := \overline{CT}(\mathtt{C}) \twoheadleftarrow \mathtt{B}'\ \mathtt{m(B\ x)}\ \{\mathtt{return\ x.disp\_m(this);}\}$
```

Note that the algorithm needs both the original CT and a new class table $\overline{CT}$, which contains the translated classes; the former is used to drive the translation, in particular for the lookup functions and for retrieving the original method definitions, while the latter is updated at every step (we use the assignment operator := to update the class table $\overline{CT}$). Thus, we start by initializing $\overline{CT}$ with a copy of the original class table CT (this way we also copy all the class fields and constructor definitions that will not be changed during the translation). Then, for every class in CT we perform the three steps that act on method names and method definitions. Note that the renaming (step 2(a)i) is always performed starting from $\overline{CT}$, thus incrementally renaming method names one by one. Step 2(a)iiA generates the disp_ methods in target classes; the branches to be generated are collected starting from the original client class in CT, using the *mtype* function that implements the copy semantics of inheritance; this is crucial to make the translation algorithm work correctly. Step 2(a)iiB adds the *entry point* branches, that have the same name of the multi methods in the original program, that forward to the disp_ methods. Note that in Steps 2(a)iiA and 2(a)iiB it is crucial to start from the original class, otherwise, the algorithm would treat also the added or renamed methods in successive steps.

Finally, we observe that the translated program $(\overline{CT}, \mathtt{e})$ differs from the original one $(CT, \mathtt{e})$ only with respect to method definitions, affecting neither the body of the original methods (i.e., the translation never acts on method bodies) nor the main expression e. The translation algorithm here defined preserves both type safety and semantics of the original programs, then it implements dynamic overloading in a type safe and correct way.

3 Conclusions and Related Work

Some object-oriented languages and calculi have been proposed to study multi-methods and dynamic overloading; we refer to [2] for an extensive discussion of these approaches. Here we list only a few recent works which are more related to the issues of the present paper.

Parasitic methods [4] are a linguistic extension of Java with asymmetric multi-methods, with the main goal of retaining modularity. The approach does not use copy semantics, the selection of the most specialized method relies on instanceof checks and consequent type casts (thus it does not perform constantly as in our solution, but essentially linearly on the number of branches), the dispatching semantics is complicated by the use of textual order of method declarations.

MultiJava [6] is a backward-compatible extension to Java supporting multi-methods and open classes. New methods in a class C can be added by defining external *method* *families* in a different compilation unit w.r.t. to the one containing C definition. The drawback of this approach is that extending, or modifying an existing method, can only be done by explicitly subclassing all affected variants and overriding the corresponding branches. This complicates the extensibility and can lead to an inconsistent distribution of code. *MultiJava* is directly compiled into Java bytecode. *Relaxed MultiJava* [9] increases flexibility of MultiJava w.r.t. overloading; however, it is obtained by allowing a function call to be ambiguous; these ambiguities are caught at class load time.

In [10] C++ is extended with open multi-methods and symmetric dispatch. Differently from our approach, open multi-methods are external to classes, as the external added methods of MultiJava.

The translation algorithm presented in this paper can be extended in two directions. Firstly, we could consider methods with more than one parameter. In this case, the translation gets more complicated, and it is the subject of an ongoing work. The second extension, which has a stronger practical impact, consists in handling multiple inheritance. Multiple inheritance introduces subtle ambiguity problems that have always been the drawback of introducing dynamic overloading in mainstream languages. From the point of view of the language, in [2] we showed how the type-checking can rule out all possible ambiguities due to multiple inheritance; then, we have only to extend the translation algorithm.

References

1. Ancona, D., Drossopoulou, S., Zucca, E.: Overloading and Inheritance. In: FOOL 8 (2001)
2. Bettini, L., Capecchi, S., Venneri, B.: Featherweight Java with Dynamic and Static Overloading. Science of Computer Programming 74(5-6), 261–278. Elsevier, Amsterdam (2009)
3. Bettini, L., Capecchi, S., Venneri, B.: Double Dispatch in C++. Software: Practice and Experience 36(6), 581–613 (2006)
4. Boyland, J., Castagna, G.: Parasitic Methods: Implementation of Multi-Methods for Java. In: Proc. of OOPSLA, pp. 66–76. ACM, New York (1997)
5. Castagna, G.: Object-Oriented Programming: A Unified Foundation. Progress in Theoretical Computer Science. Birkhauser, Basel (1997)
6. Clifton, C., Millstein, T., Leavens, G.T., Chambers, C.: MultiJava: Design rationale, compiler implementation, and applications. ACM Trans. Prog. Lang. Syst. 28(3) (May 2006)
7. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM TOPLAS 23(3), 396–450 (2001)
8. Ingalls, D.: A Simple Technique for Handling Multiple Polymorphism. In: Proc. OOPSLA, pp. 347–349. ACM Press, New York (1986)
9. Millstein, T., Reay, M., Chambers, C.: Relaxed multijava: balancing extensibility and modular typechecking. SIGPLAN Not. 38(11), 224–240 (2003)
10. Pirkelbauer, P., Solodkyy, Y., Stroustrup, B.: Open multi-methods for C++. In: GPCE, pp. 123–134. ACM, New York (2007)

Fundamental Concepts for the Structuring of Functionality into Modular Parts*

Alexander Gruler[1] and Michael Meisinger[2]

[1] Institut für Informatik, Technische Universität München, Germany
gruler@in.tum.de
[2] Calit2, University of California, San Diego, USA
mmeisinger@ucsd.edu

Abstract. Today, many software systems offer a multitude of different, user-observable functions, which in their entirety form the very complex overall system's functionality. However, practical experience shows that many question are directly related to the user-observable sub-functions. Regarding the development process, this requires to relate the entire system's functionality to its sub-functions in a formal way. In this context, decomposing and modeling the functionality in a structured way is essential. In this paper, we identify and define fundamental concepts for the structuring of a system's functionality into modular parts. We formalize these concepts using Focus, a stream-based theory for the specification of reactive systems. In particular, we define the notion of self-contained, autonomous sub-functions and introduce a canonical decomposition of functionality, inherent to the structure and nature of the functionality. Subsequently, we discuss topics of methodology that guide a modular functional decomposition. All in all, this gives a modular structuring concept for the behavior of multi-functional systems.

1 Introduction

The functionality offered by software systems is continuing to move towards the center of interest for system development. For many software-intensive systems, the overall functionality is a combination of different individual, user-observable functions which in general serve different purposes. In this context, we also speak of *multi-functional* systems; typical examples from our daily life are mobile phones and premium class automobiles. In a modern automobile, e.g. the entertainment devices and the driver assistance functions are commonly seen as different, independent sub-functions of the system "car".

When looking at a system, we can decompose its entire functionality in arbitrary ways into sub-functions. What an observer actually perceives as a standalone sub-function is very subjective and influenced by various design drivers. Essential for the construction of the entire system behavior is how to relate its individual sub-functions. Typical relations include the "is-part-of" relation, the

* This paper is available in a much more detailed version as the Technical Report [5].

F. Arbab and M. Sirjani (Eds.): FSEN 2009, LNCS 5961, pp. 463–470, 2010.

"influences"-relation and the refinement relation; e.g. the functionality of playing a CD *is part of* the entertainment function, while the driver assistance functions *influence* the movement of the car (The car does not accelerate if the Electronic Stability Control detects skids, even if the gas pedal is pushed further down). Together, these relations induce a structure on the set of functions.

The structure of the functionality is of particular interest, especially for the development of subsequent design models, such as component architectures, and of concrete implementations. Having only a loose set of unrelated sub-functions provides no guidance on how the functionalities should be implemented by components. A structured view on the functionality is important for both the consumers and the developers of a system. In particular, a model of the functionality is the link between the needs of the consumers and the construction approach of the implementers. Furthermore, a structured functionality model helps to effectively deal with challenges such as unwanted feature interactions. However, compared to the systematic development of component architectures, the act of decomposing a system's overall functionality is currently less understood. Structuring the functionality into a hierarchical way is not yet guided by conceptual or methodological principles. Besides, the meaning of proven engineering concepts such as modularity and compositionality (cf. Parnas [6] and Dijkstra [4]) is not precisely defined for the structuring of a system's functionality.

In this paper, we identify fundamental concepts for the structuring of a system's functionality and define them using a stream-based semantics based on Focus [3]. Here, individual functionalities are represented as *services*. We motivate and formally define the notion of (independent) autonomous services. We argue that functionality can be decomposed in a canonical way, where the decomposition is purely enforced by the structure and nature of the functionality itself, not by the way in which an observer perceives it. In particular, the canonical structuring is not only defined by the syntactical interface of a function, but by the (functional dependent) messages, which are communicated within a localized syntactic interface. This results in a canonical hierarchical structure for the functionality which represents a decomposition into modular, self-contained sub-functions. Based on these introduced theoretical concepts, in Section 3, we introduce and formally define further structuring principles that guide the hierarchical structuring of the behavior of multi-functional systems: We formalize the concept of a service hierarchy, which captures the decomposition of services into modular sub-services, we define the notion of a conflict between services and show how resolving conflicting services induce a natural service hierarchy. Finally, note, that this paper is available in a much more detailed version [5].

2 A Semantical Foundation for Modeling Functionality

In this section, we introduce the theoretical concepts for the modeling and hierarchical structuring of functionality. These concepts are the formal basis for a *canonical* decomposition of a function. The semantics of our concepts grounds on the Focus [3] theory for the specification and development of reactive systems.

In Focus, systems are composed of communicating components with behaviors given as input/output relations defined by total stream-processing functions. A service theory [2] exists on top of the Focus theory. A service corresponds to a part (or slice) of the (total) behavior of a component and is consequently represented as a partially defined stream-processing function. Services are offered by components and can be seen as cross-sections through the black-box behavior of a component or a system; conversely a system can be understood as the combination of all its services, cf. Figure 1(a). Services can be related to other services in various ways, e.g. a service is a *sub-service* of another (super-) service, if its functionality is completely offered by the (super-) service. Despite of the fact that knowledge of the Focus theory and its concepts is an essential requirement to understand this paper, due to space limitations we have to omit an introduction of all basic concepts of the Focus theory, here. All necessary basics are described in the extended version of this paper [5] or in [3,2].

Compared to the fundamental concepts, the concepts we introduce in this section support a systematic design methodology (cf. Section 3) which is set up for the construction of a maximally modular service hierarchy. But what is a "maximally modular" decomposition? It is a decomposition of a function into *independent* sub-services, where each sub-service encapsulates a self-contained, closed piece of functionality. We call such a decomposition *canonical* because it is only induced by the nature of the function itself, i.e. the way how the function relates inputs to outputs, respectively, and not by another, arbitrary criteria.

The fundamental idea behind a canonical decomposition is that of *autonomous service partitions*. While service slicing and projection [2,5] are technical means to allow the derivation of many possible sub-services from a given service, these sub-services do not necessarily encapsulate an independent or connected piece of functionality. In contrast, an autonomous service partition represents an independent, implied piece of functionality, i.e. it is a self-contained sub-service whose outputs are exclusively influenced by messages on its input channels.

Definition 1 (Autonomous Service Partition (ASP)). *Let $S \in \mathbb{F}[I \blacktriangleright O]$ and $S' \in \mathbb{F}[I' \blacktriangleright O']$ be two services where $(I' \blacktriangleright O')$* **subIntf** *$(I \blacktriangleright O)$, and $I', O' \neq \emptyset$. We call the service S' an* autonomous service partition (ASP) *of S if*

(i) $\forall x, \tilde{x} \in Dom(I) : \quad (x|I' = \tilde{x}|I') \implies$
$$\Big(\{y \mid \exists z : (z \in S.x) \wedge (y = z|O')\ \} = \{y \mid \exists z : (z \in S.\tilde{x}) \wedge (y = z|O')\ \}\Big)$$

(ii) and $S' = S \dagger (I' \blacktriangleright O')$ is a service slice of S .

We write $S\natural(I' \blacktriangleright O')$ to denote an ASP in $\mathbb{F}[I' \blacktriangleright O']$ which is derived from S.

According to the definition, an ASP is defined on a subset of the input channels of a service, in a way that this subset holds all necessary information to produce the outputs for the given subset of output channels. In other words, the output histories of an ASP only depend on those parts of the input histories that are not discarded by the restriction operation |, i.e. the restriction has captured and preserved all functional coherences contained in the original (unrestricted) histories. Note, that for an ASP we require that the projection contains at least one input and one output channel. There exists always an autonomous partitioning of a service *Sys* into sub-services. In some cases the only existing decomposition is the service *Sys* itself, which we then call the *trivial* ASP. In such a case the service specifies a behavior which is inherently complex, i.e. where for all outputs actually the configuration of all input channels, i.e. the entire history $\in \mathbb{H}(I)$, is necessary. Illustratively, this means that we cannot find a subset of output channels where the output histories depend on the messages of a subset of the input channels only.

An ASP may encapsulate several independent ASPs. For a canonical decomposition we are interested in minimal ASPs which are unique regarding the decomposition. A *minimal* ASP represents a sub-service that contains exactly one autonomous behavior and does not encapsulate any further non-trivial ASPs. In particular, from a minimal ASP all its input channels are essentially needed: Projecting away one more input channel destroys the property of being an ASP.

Definition 2 (Minimal and Exact Autonomous Service Partitions). *We call an ASP* $S' = S\natural(I' \blacktriangleright O')$ minimal, *if*

$$\nexists(\widetilde{I} \blacktriangleright \widetilde{O}) \quad : \quad \big((\widetilde{I} \blacktriangleright \widetilde{O})\ \mathbf{subIntf}\ (I' \blacktriangleright O')\big) \quad \wedge \quad S\natural(\widetilde{I} \blacktriangleright \widetilde{O})$$

We write $S\natural_{min}(I' \blacktriangleright O')$ *for a minimal ASP. We call a minimal ASP* $S\natural_{min}(I' \blacktriangleright O')$ exact, *if there exists no other minimal ASP* $\bar{S}$ *with the same set of input channels* I'*, but spanning more output channels, i.e. with* $\bar{S} = S\natural_{min}(I' \blacktriangleright \bar{O})$ *and* $O' \subset \bar{O}$*. We write* $S\natural_{exct}(I' \blacktriangleright O')$ *for an exact ASP.*

The concept of a minimal ASP is not yet sufficient to define a canonical decomposition, since for a given minimal ASP we can remove output channels while still preserving the property of being a minimal ASPs. The notion of an exact ASP uniquely marks that minimal ASP in such a family of ASPs which comprises the largest set of output channels. The set of all *exact* ASPs characterizes the most modular structuring of a service where all independent sub-functions are represented as individual ASPs. The following definition reflects this idea.

Definition 3 (Canonical Functional Decomposition). *Let* $S_i \in \mathbb{F}[I_i \blacktriangleright O_i], i \in \mathbb{N}$*, be services. A* canonical functional decomposition *of a service* $Sys \in \mathbb{F}[I_{Sys} \blacktriangleright O_{Sys}]$ *is the maximal set of services* $\{S_1, \ldots, S_n\}$ *such that*

(i) every service S_i *is an exact autonomous partition of* Sys*, i.e.* $\forall i \in \{1, \ldots, n\} \ : \ S_i = Sys\natural_{exct}(I_i \blacktriangleright O_i)$,

(ii) and the sets of input and output channels of the services S_i *are pairwise disjoint, i.e.* $\forall i, j \in \mathbb{N} : \quad I_i \cap I_j = \emptyset = O_i \cap O_j$ *,*

If the service Sys *is completely covered by the exact partitions* S_i*, that is if* $I_{Sys} = \biguplus_i I_i$ *and* $O_{Sys} = \biguplus_i O_i$*, we call the canonical decomposition* perfect.

Note, that the requirement (ii) in Def. 3 is implied by (i), since minimal partitions are always disjoint regarding their syntactical interfaces. Further, note that a canonical decomposition is always unique. Since we can always find an autonomous partition of a service, we can also always find a canonical decomposition, which in the most trivial case is simply a singleton set containing only a single service, namely the original service *Sys*. We call such a canonical decomposition *trivial*. A detailed example illustrating Def. 1-3 can be found in [5].

3 Methodological Integration of the Theoretical Concepts

The theoretical concepts introduced so far provide the necessary formal framework for decomposing services. However, they give no guidance of how to construct a useful hierarchical decomposition of the entire system's functionality. In the following, we discuss in a methodological context how the introduced concepts can be used to construct such a service hierarchy. Before we do so, we briefly define the concept of a (canonical) service hierarchy based on [5,1].

The fundamental idea of a service hierarchy is to decompose a service into a set of sub-services. Sub-services themselves can be further decomposed into sub-services again. This leads to a hierarchical structuring of services, the *service hierarchy*, which itself is a variant of a rooted tree where the nodes represent services and the edges represent the sub-service relation.

Definition 4 (Service Hierarchy). *Let $H = (V, E)$ be a rooted tree, $spec : V \to \mathbb{F}$ a function which associates a service (specification) to every node of H. We call the tuple $\big((V, E), spec\big)$ a* service hierarchy*, if*

(i) every child-node is a sub-service of its parent node, i.e. $\forall (u, v) \in E \;:\; spec(v) \prec_{sub} spec(u)$,

(ii) and the syntactical interface of any compound service is determined by the interfaces of its children, i.e. $\forall u \in V,\ spec(u) \in \mathbb{F}[I \blacktriangleright O]$: $\big(I = \bigcup_{v:\ (u,v)\in E} In(spec(v))\big) \;\wedge\; \big(O = \bigcup_{v:\ (u,v)\in E} Out(spec(v))\big)$

The ruling principle in a service hierarchy is that the behavior (I/O-relation) of any compound service can be completely derived from the behaviors of all of its sub-services. This allows to split up a service into a set of sub-services, where on the one hand both the semantical and syntactical interfaces of the super-service are determined by its sub-services. On the other hand, a super-service cannot have an "additional" functionality which is not part of its sub-services.

Definition 5 (Canonical Service Hierarchy). *Let $SD = \big((V, E), spec\big)$ be a service hierarchy. We call SD* canonical *if for every service s, which is not a leaf node, the set of its children $children(\rho(s))$ is its perfect canonical decomposition.*

A canonical service hierarchy is a hierarchy with depth 1. It represents the most modular form of how a functionality can be structured in direct sub-functions. It is a decomposition into purely self-contained, independent sub-services which cover the entire (syntactical and semantical) interface of their super-service. The sub-services of such a canonical hierarchy represent functionalities which cannot be further decomposed into autonomous parts.

3.1 Hierarchical Structuring of Services

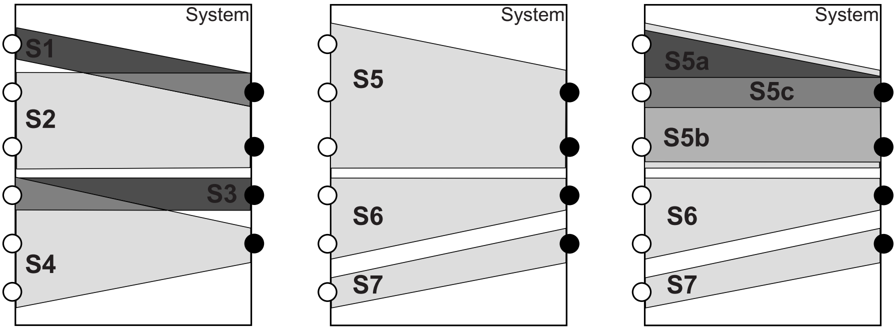
Methodologically, a service hierarchy can be the result of different structuring strategies following different approaches. In the following, we differentiate between two scenarios: (1) The decomposition of a very complex service into a service hierarchy according to certain criteria, and (2) the construction of a service hierarchy from a flat set of (unrelated) services (representing use cases).

Decomposition of a Service into a Modular Service Hierarchy An important design driver for the transition to subsequent design models is to structure the functionality, represented by the set of services, in a maximally modular way. Here, with "modular", we mean decomposition of the functionality offered by a system—represented by a single, possibly very complex service S—into smaller, self-contained, non-overlapping parts that are functionally independent and at the same time cover a maximal part of the behavior of S.

The canonical decomposition (cf. Def. 3) of a service embodies this idea. It represents a partitioning of a service S into disjoint, modular parts $\{S_1, \ldots, S_n\}$, where each partition $S_i, i \in \mathbb{N}$, encapsulates a separate piece of behavior within a precisely localized syntactical interface. In particular, the canonical decomposition is *perfect* if the partitions cover the entire behavior of the original service S. Figure 1(b) shows a perfect canonical decomposition into three services.

By adding all the service partitions S_i as sub-services of S, we get a canonical service hierarchy (cf. Def. 5) of S. Because the S_i are a perfect canonical decomposition of S, the required property of a service hierarchy that the behavior of any compound service must always be completely defined by the behaviors of its sub-services, is automatically guaranteed.

A (non-trivial) perfect canonical decomposition does not always exist: In some cases where we find a canonical decomposition, the contained autonomous service partitions do not completely cover the functionality required by their super-service. In many cases we cannot even find a non-trivial canonical decomposition, since the functional dependencies of the service to be decomposed are too complex. In such a case, a decomposition will always result in a set of sub-services with "overlapping" functionality, i.e. is *not* autonomous in the sense of Definition 1. For more details concerning a decomposition methodology see the corresponding part in [5].



(a) An arbitrary decomposition into sub-services.

(b) Perfect canonical decomposition into exact autonomous partitions.

(c) Further hierarchical decomposition of the autonomous partition S5.

Fig. 1. Different decompositions of the black-box behavior of a component. A white circle denotes an input channel, a black circle denotes an output channel.

Hierarchical Combination of Services Induced by Conflicts During the early phases of system development, services provide an adequate way of formalizing the requirements posed to a system. From this point of view, a service can be seen as the formalization of a use-case, restricting the set of possible implementations of the system. Specifying the functionality of a system in this way usually results in a set of unrelated services as illustrated in Figure 1(a): Here, the services may overlap in arbitrary ways, i.e. some services depend on common input channels, while other services specify requirements on the same output channels.

The resulting system exhibits the behavior of all these services, i.e. it has to meet all service specifications in parallel. However, the services can be conflicting or even contradictory, in particular during early phases of RE. Here, a *conflict* is any contradicting behavior resulting from the combination of two modularly specified services that are (direct or indirect) sub-services of the same super-service. More precisely, two services are conflicting if both services produce for identical input history on common input channels different output histories on at least one common output channel.

Definition 6 (Conflicts Between Services). *Let $S_1 \in \mathbb{F}[I_1 \blacktriangleright O_1]$, $S_2 \in \mathbb{F}[I_2 \blacktriangleright O_2]$ be two services, $I = I_1 \cap I_2$ a common subset of input channels, and $O = O_1 \cap O_2$ a common subset of output channels of S_1 and S_2 . We call the services S_1 and S_2* conflicting*, if*

$$\begin{aligned} &\exists x_1 \in \mathit{Dom}(S_1),\ x_2 \in \mathit{Dom}(S_2),\ x \in \mathbb{H}(I): \\ &\quad \big(x_1|I = x = x_2|I\big) \quad \wedge \quad \big(S_1 \dagger (I \blacktriangleright O).x \neq S_2 \dagger (I \blacktriangleright O).x\big) \end{aligned}$$

Otherwise we call the services conflict free*. The histories x are called* conflicts*. We write $\mathit{Conflicts}(S_1, S_2)$ to denote the set of all conflicting input histories x between the services S_1 and S_2.*

Because the system has to provide the functionality specified by all individual services, we have to resolve conflicts within the set of services in order to construct a valid system. One way to resolve a conflict between two services is to prioritize one service over the others. However, prioritization of services already hints at which services depend on one another in the sense that they describe a related piece of functionality. More precisely, the act of prioritization of services induces a "natural" hierarchical structure on the set of services.

From a methodological point of view, this provides guidance of how to combine services in a hierarchy in order to yield compound services: Given two conflicting services $S_1 = (I_1 \blacktriangleright O_1)$ and $S_2 = (I_2 \blacktriangleright O_2)$, we specify a new service $R = (I_1 \cup I_2 \blacktriangleright O_1 \cup O_2)$, called the *resolving service*, where $\mathit{Dom}(R) = \mathit{Conflicts}(S_1, S_2)$. The behavior of R needs to be specified manually by the system designer, implementing the design decisions how the conflicts are resolved. The semantical interpretation of the service hierarchy has to ensure that all input histories in $\mathit{Dom}(R)$ are processed by the resolving service R only. Then, the services S_1, S_2 and R can be combined forming a new super-service S, since the behavior of S is completely defined by the services S_1, S_2 and R.

Figure 1(c) illustrates the composition of services in this way: The services $S5a, S5b$ and $S5c$ are composed forming the service $S5$ shown in Figure 1(b). The structuring of $S5$ in this way yields a representation that explicitly distinguishes between its modular ($S5a$ and $S5b$) and overlapping parts ($S5c$), which encapsulates the conflicting input histories. Regarding feature interaction, this has the advantage to leave the modular specification of the conflicting services untouched and to make the resolution of the conflict explicit in a specific service.

4 Conclusion

With the increasing complexity of software-intensive systems, their offered functionality moves in the center of interest for an effective development. In particular, the construction of multi-functional, distributed systems with their complex functional dependencies and interactions benefits from functionality oriented design and implementation strategies. We have introduced and defined fundamental concepts for the hierarchical structuring of functionality. Based on the concept of sub-service, we have formalized the general concept of functional hierarchy, and in particular of its canonical form. The functionality of a system can be structured in different ways. One fundamental way —especially valuable for the transition to subsequent design models or modular verification— is to decompose the functionality into independent, self-contained parts. We have formalized this idea introducing the concept of autonomous services. A structuring into autonomous services represents a canonical decomposition implementing the idea of functional modularity. For non-autonomous services, we have defined the notion of a conflict. Based on this notion, we can characterize and represent conflicting functionalities as a precisely defined set of related services, where the part of the functionality which is not involved in the conflict, and the conflicting part, are explicitly modeled. We have placed the introduced concepts into a methodological context and outlined their application for constructing a service hierarchy. Note that due to space limitations, we had to omit essential concepts and definitions which are described in more detail in [5].

References

1. Broy, M.: Two sides of structuring multi-functional software systems: Function hierarchy and component architecture. In: Kim, H.-K., et al. (eds.) SERA 2007, pp. 3–10. IEEE Computer Society, Los Alamitos (2007)
2. Broy, M., Krüger, I., Meisinger, M.: A formal model of services. ACM Transactions on Software Engineering Methodology (TOSEM) 16(1) (2007)
3. Broy, M., Stølen, K.: Specification and Development of Interactive Systems - Focus on Streams, Interfaces and Refinement. Springer, New York (2001)
4. Dijkstra, E.W.: Ch. I: Notes on structured programming. Academic Press, London (1972)
5. Gruler, A., Meisinger, M.: Hierarchical decomposition of multi-functional systems. Technical Report TUM-I0901, Technische Universität München (2009)
6. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM 15(12), 1053–1058 (1972)

Author Index