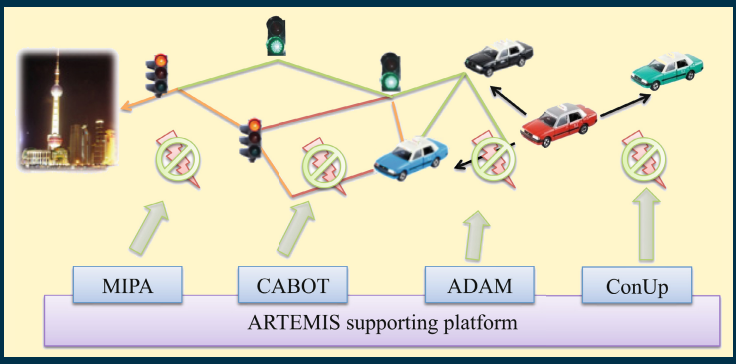
Festschrift

LNCS 8051

Zhiming Liu
Jim Woodcock
Huibiao Zhu (Eds.)

Theories of Programming and Formal Methods

Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday



Springer

Lecture Notes in Computer Science 8051

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

Zhiming Liu Jim Woodcock Huibiao Zhu (Eds.)

Theories of Programming and Formal Methods

Essays Dedicated to Jifeng He
on the Occasion of His 70th Birthday

Springer

Volume Editors

Zhiming Liu
United Nations University
International Institute for Software Technology
P.O. Box 3058, Macau, China
E-mail: z.liu@iist.unu.edu

Jim Woodcock
University of York, Department of Computer Science
Deramore Lane, York YO10 5GH, UK
E-mail: jim@cs.york.ac.uk

Huibiao Zhu
East China Normal University, Software Engineering Institute
3663 Zhongshan Road (North), Shanghai 200062, China
E-mail: hbzhu@sei.ecnu.edu.cn

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-39697-7 e-ISBN 978-3-642-39698-4
DOI 10.1007/978-3-642-39698-4
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013943015

CR Subject Classification (1998): F.3, D.2.4, D.2, F.1, F.4, D.3, I.6

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

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

Printed on acid-free paper

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

Jifeng He

Foreword

Jifeng He is an outstanding computer scientist. He was born on August 5, 1943, in Shanghai, China. In his long academic career, he has made significant and wide-ranging contributions to the theories of programming and formal software engineering methods. To celebrate his 70th birthday, we present three LNCS volumes in his honor.

- *Theories of Programming and Formal Methods. Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday.* Papers presented at a symposium held in Shanghai, September 1–3, 2013. LNCS volume 8051, Springer 2013.
- *Unifying Theories of Programming and Formal Engineering Methods.* International Training School on Software Engineering, Shanghai, China, August 26–30, 2013. Advanced Lectures, LNCS volume 8050, Springer 2013.
- Theoretical Aspects of Computing – ICTAC 2013. The 10th International Colloquium, Shanghai, China, September 4–6, 2013. Proceedings, LNCS volume 8049, Springer 2013.

He Jifeng is known for his seminal work in the theories of programming and formal methods for software engineering. He is particularly associated with *Unifying Theories of Programming (UTP)*, the theory of data refinement and the laws of programming, and the rCOS formal method for object and component system construction. His book on UTP with Tony Hoare has been widely read and followed by a large number of researchers, and it has been used in many postgraduate courses. He was a senior researcher at Oxford during 1984-1998, and then a senior research fellow at the United Nations University International Institute for Software Technology (UNU-IIST) in Macau during 1998-2005. He has been a professor and is currently the Dean of the Institute of Software Engineering at East China Normal University, Shanghai, China. He was a founder of the International Conference of Formal Engineering Methods (ICEFM), the International Colloquium on Theoretical Aspects of Computing (ICTAC), and the International Symposium on Theoretical Aspects of Software Engineering (TASE). In 2005, He Jifeng was elected as an academician of the Chinese Academy of Sciences. He also received an honorary doctorate from the University of York. He has won a number of prestigious science and technology awards, including the second prize of the Natural Science Award from the State Council of China, the first prize of the Natural Science Award from the Ministry of Education of China, the first prize of Technology Innovation from the Ministry of Electronic Industry, and a number awards from Shanghai government.

We, the three organizers of the celebration events, have all worked with He Jifeng. We thank him for his years of generous, wise advice to us and to his many other colleagues, students, and friends. He has been constantly energetic, inspiring, enthusiastic, and encouraging.

We wish him a happy birthday.

June 2013

Zhiming Liu
Jim Woodcock
Huibiao Zhu

Organization

Program Chairs

Zhiming Liu	UNU-IIST, Macau SAR, China
Jim Woodcock	University of York, UK
Huibiao Zhu	East China Normal University, China

Local Organization

Mingsong Chen, Jian Guo, Xiao Liu, Geguang Pu, Fu Song, Min Zhang
East China Normal University

Table of Contents

Set-Theoretic Models of Computations

Jean-Raymond Abrial

Marseille, France
jrabrial@neuf.fr

Abstract. The purpose of this paper is to present some set-theoretic *models of computation*. This topic and its usefulness are clearly related to those presented in the book by Hoare and He: "*Unifying Theories of Programming*" [12]. However, we prefer to use here the term "computation" to that of "programming" as our purpose is not so much to unify various ways of programming (using different programming languages) but rather to see how various mechanical computation paradigms (be they sequential, distributed, parallel, and so on) can be given a unified mathematical theory. Our purpose is also to study how these computations can be specified and then developed by means of refinements and proofs.

1 Introduction

This study is clearly not immediately related to any application. Instead we definitely and only consider the *fundamental mathematical structure* modeling mechanical computations. As pointed out in [12] and in other scientific disciplines as well, it appears to be very important to develop some mathematical models for our main concept, that is in our case that of *computation*. In doing so, we intend to be able to understand not only the well known existing mechanisms at work in computations today but also those we might encounter in the future.

This topic is by no means new. In fact, a variety of computational models have been proposed in the literature. This is carefully reviewed by Nelson in [19]. He made a clear distinction between models dealing with relations on predicate [13], relations on states [14], predicate transformers [5], or simply predicates [12].

The approach presented here deals with *relations on states* and with *set transformers* (the set-theoretic equivalent to predicate transformers). For this, I use set theory rather than predicate calculus. To the best of my knowledge, it has not been done so far systematically in this way. The reason why I favor set theory over predicate calculus as a medium for such a theoretical development is certainly one of personal taste: I prefer to quantify over sets (or better, over all subsets of a certain set) than over predicates, and I think that set complementation is more convenient than predicate negation for theoretical developments and in mechanized proofs as well.

The paper is organized as follows. In section 2, two *equivalent* models of finite computations are recalled: the forward model and the backward model. Then a number of elementary combinators are proposed. In section 3 we study infinite computations and iterations. This will give us enough material to develop the

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 1–22, 2013.

notion of modalities. We introduce then hiding and projection in section 4 and refinement in section 5. The last section is entirely devoted to the mapping of these approaches to several specific practical *computation paradigms*. Finally, two appendices are proposed to help the reader remembering some well known mathematical results that are use throughout this paper. As a consequence, no complicated prerequisites, besides elementary predicate logic and set theory, are required to read this paper.

2 Finite Computations

To formalize these approaches, we start by defining a set A (supposed to be non-empty) corresponding to the possible values of all the variables of a computation: it is usually a cartesian product together with some additional properties.

2.1 The Forward Approach

We abstract a computation as a relation r built on the set A:

$$r \subseteq A \times A \tag{1}$$

In order to take account of the requirement of a finite computation, we consider a set p representing the set of states on which an execution of the future computation can be started with the *full guarantee to terminate in a finite time*:

$$p \subseteq A \tag{2}$$

Note that the set p is by no means the *domain* of the relation r. In order to emphasize that we do not care what the computation does if started outside p (it can give some strange results or loop for ever), we simply state that each point lying within $\overline{p}$ (the complement of p with respect to A, that is $A \setminus p$) is connected to any point of A through r. This yields the following characteristic property:

$$\overline{p} \times A \subseteq r \tag{3}$$

2.2 The Backward Approach

We now follow the famous backward approach of Dijkstra [5] [6]: we suppose that the computation has *successfully* taken place and terminates in an "after" state belonging to a certain subset q of A. We wonder then what the corresponding *largest* "before" set $F(q)$ is. According to this point of view, a computation can thus be modeled by the set function F transforming the after set q into the before set $F(q)$. This yields the following:

$$F \in \mathbb{P}(A) \to \mathbb{P}(A) \tag{4}$$

We call the set function F a *set transformer*. If we terminate in the set $q1 \cap q2$ then we must have started in the intersection of the corresponding before sets, that is:

$$F(q1 \cap q2) \;=\; F(q1) \cap F(q2) \tag{5}$$

This characteristic property of F, the *conjunctivity* property, can be generalized to a set of sets. The conjunctivity property has the easy consequence that the set transformer F is monotonic:

$$q1 \subseteq q2 \;\Rightarrow\; F(q1) \subseteq F(q2) \tag{6}$$

2.3 Equivalence of the Two Approaches

It can be shown that the two previous approaches are equivalent. In other words, it is possible to deduce one from the other and vice-versa. Given a relation r and set p as in section 2.1, one can define the set transformer F as follows:

$$F(q) \;=\; p \;\cap\; \overline{r^{-1}[\overline{q}]} \tag{7}$$

This result can be made a little more precise as follows:

$$F(q) \;=\; \begin{cases} \overline{r^{-1}[\overline{q}]} & \text{if} \quad q \neq A \\ \\ p & \text{if} \quad q = A \end{cases} \tag{8}$$

From this, it is easy to prove the conjunctivity property (5). Conversely, given a set transformer F as in section 2.2, one can define a set p and a relation r as follows:

$$\begin{cases} p = F(A) \\ \\ r = \{x \mapsto x' \,|\, x \in \overline{F(\overline{\{x'\}})}\} \end{cases} \tag{9}$$

From this, it is easy to prove the characteristic property (3).

2.4 Combinators

From the previous forward and backward models associated with a finite computation, we now introduce a number of *elementary combinators* in the following table. Note that some of these combinators do not exist as such in programming languages but they can be put together to form more elaborate classical combinators (this will be done in Section 6). Here is a short description of these elementary combinators:

1. The *deterministic assignment* ($x := E(x)$) is present in all imperative programming languages.
2. The *non-deterministic assignment* ($x :\in B(x)$) and the *post-conditioning* ($x :|\, a(x, x')$) are not present in programming languages. They are specification concepts that owe to be refined later during the design of a program.

3. The *sequence* $(S1\,;S2))$ is present in all imperative programming languages. We consider that such computations are not specification concepts since they define the way things are arranged in order to achieve a certain goal.
4. The *bounded choice* $(S1 \sqcap S2)$ and the *unbounded choice* $(\sqcap_{z\in u} S_z)$ allow us to define a non-deterministic choice between two or more computations. It does not exist as such in programming languages. This is rather a specification concept allowing us to express that our future computation should behave in one way or another, the choice being made by refinement during the design of the computation.
5. The *conjunct* $(S1 \sqcup S2)$ allows us to specify a computation by means of two or more properties. It does not exist in programming languages . Notice that the same purpose can be obtained by using post-conditioning.
6. The *parallelism* $(S1||S2)$ allows us to define the simultaneous execution of two or more computations. During a program design it will be replaced by a sequencing statement.
7. The *guarding* $(b(x) \Longrightarrow S)$ has the effect of restricting the relation associated with a computation. It does not exist as such in programming languages but it is used to express more classical statements such as conditional and loop (see Sections 6.1 and 6.2).
8. The *pre-conditioning* $(b(x) \mid S)$ has the effect of forcing a pre-condition to hold before executing the associated computation. If it is not the case, then the overall computation aborts.
9. The *skip computation* does not do anything. It is used implicitly in programming languages.
10. The *magic computation*, the *any choice computation*, and the *abort computation* are artificial concepts used in theoretical developments.

In the following table, S, $S1$, and $S2$ are computations. Then (r, p), $(r1, p1)$, and $(r2, p2)$ are the corresponding forward models. Finally F, $F1$, and $F2$ are the corresponding backward models.

Combinator	Relation Pre-condition	Set Transformer Value at q
Deterministic Assignment $x := E(x)$	$\{x \mapsto x' \mid x' = E(x)\}$ A	$\{x \mid E(x) \in q\}$
Non deterministic Assignment $x :\in B(x)$	$\{x \mapsto x' \mid x' \in B(x)\}$ A	$\{x \mid B(x) \subseteq q\}$
Post-Conditioning $x :\mid a(x, x')$	$\{x \mapsto x' \mid a(x, x')\}$ A	$\{x \mid \forall x' \cdot a(x, x') \Rightarrow x' \in q\}$

Sequence $S1; S2$	$(\overline{p1} \times A) \cup (r2 \circ r1)$ $p1 \cap \overline{r1^{-1}[\overline{p2}]}$	$F1(F2(q))$
Bounded Choice $S1 \sqcap S2$	$r1 \cup r2$ $p1 \cap p2$	$F1(q) \cap F2(q)$
Unbounded Choice $\sqcap_{z \in u} S_z$	$\bigcup_{z \in u} r_z$ $\bigcap_{z \in u} p_z$	$\bigcap_{z \in u} F_z(q)$
Conjunct $S1 \sqcup S2$	$r1 \cap r2$ $p1 \cup p2$	(11)
Parallelism $S1 \| \| S2$	(10) $p1 \times p2$	(12)
Guarding $b(x) \Longrightarrow S$	$(\{x \mid b(x)\} \times A) \cap r$ $\overline{\{x \mid b(x)\}} \cup p$	$\overline{\{x \mid b(x)\}} \cup F(q)$
Pre-conditioning $b(x) \mid S$	$(\overline{\{x \mid b(x)\}} \times A) \cup r$ $\{x \mid b(x)\} \cap p$	$\{x \mid b(x)\} \cap F(q)$
skip	id A	q
magic	$\varnothing$ A	A
choice	$A \times A$ A	$\begin{cases} \varnothing & \text{if } q \neq A \\ A & \text{if } q = A \end{cases}$
abort	$A \times A$ $\varnothing$	$\varnothing$

$$(\overline{(p1 \times p2)} \times (A1 \times A2)) \ \cup \ (r1 \parallel r2) \tag{10}$$

$$\begin{cases} \bigcap_{x' \in \overline{q}} F1(\overline{\{x'\}}) \cup F2(\overline{\{x'\}}) & \text{if } q \neq A \\ F1(A) \cup F2(A) & \text{if } q = A \end{cases} \tag{11}$$

$$\begin{cases} \bigcap_{x' \mapsto y' \in \overline{q}} F1(\overline{\{x'\}}) \cup F2(\overline{\{y'\}}) & \text{if } q \neq A1 \times A2 \\ F1(A1) \times F2(A2) & \text{if } q = A1 \times A2 \end{cases} \tag{12}$$

It is not difficult to verify that all combinators presented in the above table result in conjunctive set transformers and also that the forward models obey their characteristic property.

3 Iteration and Infinite Computations

In this section[1], we study iteration, in the backward and then in the forward approach. Usually, iteration is studied under the most classical form of a while loop:

while G **do** S **end**

where G is a boolean expression (the guard of the loop) and S is a non-guarded computation (the body of the loop).The unfolding of the while loop yields:

$$\begin{array}{lcl} & & \textbf{if } G \textbf{ then} \\ & & \quad S; \textbf{while } G \textbf{ do } S \textbf{ end} \\ \textbf{while } G \textbf{ do } S \textbf{ end} & = & \textbf{else} \\ & & \quad \mathsf{skip} \\ & & \textbf{end} \end{array} \tag{13}$$

In what follows, we do not formalize iteration like this because it is too complicated, we rather use a more abstract form, $S^{\triangledown}$, called the *abstract iteration*, unfolded as follows:

$$S^{\triangledown} \;=\; \mathsf{skip} \sqcap (S\,;S^{\triangledown}) \tag{14}$$

The while loop can then be defined with various combinators of section 2.4:

$$\textbf{while } G \textbf{ do } S \textbf{ end} \;\widehat{=}\; (G \Longrightarrow S)^{\triangledown}\,;(\neg\, G \Longrightarrow \mathsf{skip})$$

3.1 Abstract Iteration in the Backward Approach

As we have just seen, abstract iteration obeys the following equation:

$$S^{\triangledown} \;=\; \mathsf{skip} \sqcap (S\,;S^{\triangledown}) \tag{15}$$

Translating this to the backward set transformer by means of the combinators of section 2.4, yields the following where F is the conjunctive set transformer corresponding to S and $F^{\triangledown}$ is that corresponding to $S^{\triangledown}$:

$$F^{\triangledown}(q) \;=\; q \cap F(F^{\triangledown}(q)) \tag{16}$$

[1] All proofs mentioned, but not necessarily proven, in this section have been mechanically verified by the theorem prover of the Rodin Platform [20].

Given a set transformer G, let $p \,|\, G$ be the set transformer where:

$$(p \,|\, G)(k) \;\widehat{=}\; p \cap G(k) \tag{17}$$

Then equality (16) yields:

$$F^{\,\triangledown}(q) \;=\; (q \,|\, F)(F^{\,\triangledown}(q))$$

As can be seen, $F^{\,\triangledown}(q)$ appears to be a *fixpoint* of the set function $q \,|\, F$. We can then define $F^{\,\triangledown}(q)$ as follows by means of a least fixpoint:

$$F^{\,\triangledown}(q) \;\widehat{=}\; \mathsf{fix}(q \,|\, F) \tag{18}$$

We also define another combinator corresponding to the greatest fixpoint[2]:

$$F^{\,\triangle}(q) \;\widehat{=}\; \mathsf{FIX}(q \,|\, F) \tag{19}$$

Of course, we have to prove that these combinators are conjunctive. But, before that, we present the relationship between the two. More precisely, we prove the following:

$$F^{\triangledown} \;=\; \mathsf{fix}(F) \,|\, F^{\,\triangle} \tag{20}$$

We first prove $F^{\,\triangledown}(q) \subseteq \mathsf{fix}(F) \,\cap\, F^{\,\triangle}(q)$

Proof

$F^{\,\triangledown}(q) \subseteq \mathsf{fix}(F) \,\cap\, F^{\,\triangle}(q)$
$\Leftrightarrow$ According to Definition (18)
$\mathsf{fix}(q \,|\, F) \subseteq \mathsf{fix}(F) \,\cap\, F^{\,\triangle}(q)$
$\Leftarrow$ According to Theorem 1 of **Appendix 1**
$(q \,|\, F)(\mathsf{fix}(F) \,\cap\, F^{\,\triangle}(q)) \subseteq \mathsf{fix}(F) \,\cap\, F^{\,\triangle}(q)$
$\Leftrightarrow$ According to (5) and Definition (17)
$q \cap F(\mathsf{fix}(F)) \,\cap\, F(F^{\,\triangle}(q)) \subseteq \mathsf{fix}(F) \,\cap\, F^{\,\triangle}(q)$
$\Leftrightarrow$ According to Definition (17)
$F(\mathsf{fix}(F)) \,\cap\, (q \,|\, F)(F^{\,\triangle}(q)) \subseteq \mathsf{fix}(F) \,\cap\, F^{\,\triangle}(q)$
$\Leftrightarrow$ According to (18) and (19)
$F(\mathsf{fix}(F)) \,\cap\, (q \,|\, F)(\mathsf{FIX}(q|F)) \subseteq \mathsf{fix}(F) \,\cap\, \mathsf{FIX}(q|F)$
$\Leftrightarrow$ According to Theorem 3 of **Appendix 1**
$\mathsf{fix}(F) \,\cap\, \mathsf{FIX}(q|F) \subseteq \mathsf{fix}(F) \,\cap\, \mathsf{FIX}(q|F)$

End of Proof

We then prove the reverse containment, that is $\mathsf{fix}(F) \,\cap\, F^{\,\triangle}(q) \;\subseteq\; F^{\,\triangledown}(q)$

[2] Least and greatest fixpoints definitions and properties are recalled in Appendix 1.

Proof

$$
\begin{array}{ll}
& \mathsf{fix}(F) \ \cap \ F^{\,\vartriangle}(q) \ \subseteq \ F^{\,\triangledown}(q) \\
\Leftrightarrow & \\
& \mathsf{fix}(F) \ \subseteq \ F^{\,\triangledown}(q) \cup \overline{F^{\,\vartriangle}(q)} \\
\Leftarrow & \text{According to Theorem 1 of } \textbf{Appendix 1} \\
& F(F^{\,\triangledown}(q) \cup \overline{F^{\,\vartriangle}(q)}) \ \subseteq \ F^{\,\triangledown}(q) \cup \overline{F^{\,\vartriangle}(q)} \\
\Leftrightarrow & \\
& F(F^{\,\triangledown}(q) \cup \overline{F^{\,\vartriangle}(q)}) \cap F^{\,\vartriangle}(q) \ \subseteq \ F^{\,\triangledown}(q) \\
\Leftrightarrow & \text{According to Definition (19)} \\
& F(F^{\,\triangledown}(q) \cup \overline{F^{\,\vartriangle}(q)}) \cap \mathsf{FIX}(q \,|\, F) \ \subseteq \ F^{\,\triangledown}(q) \\
\Leftrightarrow & \text{According to Theorem 2 of } \textbf{Appendix 1} \\
& F(F^{\,\triangledown}(q) \cup \overline{F^{\,\vartriangle}(q)}) \cap (q \,|\, F)(\mathsf{FIX}(q \,|\, F)) \ \subseteq \ F^{\,\triangledown}(q) \\
\Leftrightarrow & \text{According to Definitions (18) and (19)} \\
& F(F^{\,\triangledown}(q) \cup \overline{F^{\,\vartriangle}(q)}) \cap q \cap F(F^{\,\vartriangle}(q)) \ \subseteq \ F^{\,\triangledown}(q) \\
\Leftrightarrow & \text{According to (5)} \\
& q \cap F(F^{\,\triangledown}(q) \cap F^{\,\vartriangle}(q)) \ \subseteq \ F^{\,\triangledown}(q) \\
\Leftrightarrow & \text{According to (18), (19), and Theorem 5 of } \textbf{Appendix 1} \\
& (q|F)(F^{\,\triangledown}(q)) \ \subseteq \ F^{\,\triangledown}(q) \\
\Leftrightarrow & \text{According to (18) and Theorem 6 of } \textbf{Appendix 1} \\
& F^{\,\triangledown}(q) \ \subseteq \ F^{\,\triangledown}(q)
\end{array}
$$

End of Proof

It remains now for us to prove conjunctivity. It is obviously sufficient to prove that of $F^{\,\vartriangle}$ since that of $F^{\,\triangledown}$ can then be easily deduced. This proof is left to the reader.

3.2 Abstract Iteration in the Forward Approach

As for the backward approach in the previous section, we start from the following:

$$S^{\,\triangledown} \ = \ \mathsf{skip} \ \sqcap \ (S \,;\, S^{\,\triangledown}) \tag{21}$$

We want to define the pre-condition set, $p^{\triangledown}$, and the relation, $r^{\triangledown}$, associated with $S^{\triangledown}$. We suppose that the set p and relation r correspond to the computation S.

Pre-condition Set of the Abstract Iteration. From (21) and section 2.4, we deduce:

$$p^{\triangledown} \ = \ p \cap \overline{r^{-1}[\overline{p^{\triangledown}}]} \tag{22}$$

We can then define $p^{\triangledown}$ as follows:

$$p^{\triangledown} \ \widehat{=} \ \mathsf{fix}(\lambda q \cdot p \cap \overline{r^{-1}[\overline{q}]}) \tag{23}$$

Note that this definition is coherent with (9), (18), and (7), since we have:

$$p^{\triangledown} \ = \ F^{\triangledown}(A) \ = \ \mathsf{fix}(A \,|\, F) \ = \ \mathsf{fix}(F) \ = \ \mathsf{fix}(\lambda q \cdot p \cap \overline{r^{-1}[\overline{q}]}) \tag{24}$$

Taking the complement of both sides in (22) yields:

$$\overline{p^\nabla} \;=\; \overline{p} \cup r^{-1}[\overline{p^\nabla}] \tag{25}$$

This interesting result (25) can be given the following operational explanation: when a point x belongs to $\overline{p^\nabla}$, the abstract iteration aborts because either the body of the abstract iteration aborts ($x \in \overline{p}$) or there exists a point x', also in $\overline{p^\nabla}$, such that $x \mapsto x' \in r$ ($x \in r^{-1}[\overline{p^\nabla}]$). In this case, we clearly loop for ever. Now we would like to prove that the relation r restricted to p^∇ is *well-founded*. For this, according to the definition of a well-founded relation recalled in Appendix 2, we have to prove the following:

$$\forall l \cdot l \subseteq (p^\nabla \lhd r)^{-1}[l] \;\Rightarrow\; l = \varnothing \tag{26}$$

According to the definition (23), that is:

$$p^\nabla \;\widehat{=}\; \mathsf{fix}(\lambda q \cdot p \cap \overline{r^{-1}[\overline{q}]})$$

We can thus replace p^∇ in what follows by expanding its definition, yielding;

$$p^\nabla \;=\; \bigcap\{q \mid p \cap \overline{r^{-1}[\overline{q}]} \subseteq q\}$$

We assume $l \subseteq (p^\nabla \lhd r)^{-1}[l]$ and we have to prove $l = \varnothing$. The proof is by contradiction. We assume thus $x \in l$ for some x and we want to derive a contradiction. The proof is left to the reader.

Relation of the Abstract Iteration. By translating equation (21) in terms of the relations r and r^∇ we obtain the following fixpoint equation:

$$r^\nabla \;=\; \mathrm{id} \;\cup\; (r \,;\, r^\nabla) \tag{27}$$

We define then r^∇ as follows:

$$r^\nabla \;\widehat{=}\; \mathsf{FIX}(\lambda s \cdot \mathrm{id} \;\cup\; (r \,;\, s)) \tag{28}$$

As for p^∇, we can prove that this definition of r^∇ is coherent with that of F^∇ in section 3.1. More precisely, according to (9), we must have:

$$r^\nabla = \{x \mapsto x' \mid x \in \overline{F^\nabla(\overline{\{x'\}})}\} \tag{29}$$

So, according to (28) and (18), we must prove the following:

$$\mathsf{FIX}(\lambda s \cdot \mathrm{id} \;\cup\; (r \,;\, s)) = \{x \mapsto x' \mid x \in \overline{\mathsf{fix}(\overline{\{x'\}} \mid F)}\} \tag{30}$$

We first decompose the two members of (30). First the left hand side:

$$\begin{array}{l} x \mapsto y \in \mathsf{FIX}(\lambda s \cdot \mathrm{id} \cup (r \,;\, s)) \\ \Leftrightarrow \\ x \mapsto y \in \bigcup\{s \mid s \subseteq \mathrm{id} \cup (r \,;\, s)\} \\ \Leftrightarrow \\ \exists s \cdot s \subseteq \mathrm{id} \cup (r \,;\, s) \;\wedge x \mapsto y \in s \end{array}$$

Then the right hand side:

$$
\begin{array}{ll}
& x \mapsto y \in \{x \mapsto x' \mid x \in \overline{\mathsf{fix}(\overline{\{x'\}} \mid F)}\} \\
\Leftrightarrow & \\
& x \in \overline{\mathsf{fix}(\overline{\{y\}} \mid F)} \\
\Leftrightarrow & \qquad\qquad \text{According to Theorem 6 of } \textbf{Appendix 1} \\
& x \in \mathsf{FIX}(\overbrace{\overline{\{y\}}}^{\sim} \mid F) \\
\Leftrightarrow & \qquad\qquad \text{According to Theorem 7 of } \textbf{Appendix 1} \\
& x \in \mathsf{FIX}(\lambda q \cdot \{y\} \cup \overline{F(\overline{q})}) \\
\Leftrightarrow & \\
& x \in \mathsf{FIX}(\lambda q \cdot \{y\} \cup r^{-1}[q]) \\
\Leftrightarrow & \\
& x \in \bigcup\{q \mid q \subseteq \{y\} \cup r^{-1}[q]\} \\
\Leftrightarrow & \\
& \exists q \cdot q \subseteq \{y\} \cup r^{-1}[q] \quad \wedge \ x \in q
\end{array}
$$

We have thus to prove the following:

$$(\exists s \cdot s \subseteq \mathrm{id} \cup (r\,;s) \ \wedge x \mapsto y \in s) \ \Leftrightarrow \ (\exists q \cdot q \subseteq \{y\} \cup r^{-1}[q] \quad \wedge \ x \in q)$$

The proof is left to the reader. By using the property (20) (that is $F^{\triangledown} = \mathsf{fix}(F) \mid F^{\,\triangle}$), we can also prove the following:

$$r^{\triangledown} \ = \ (\overline{p^{\triangledown}} \times A) \ \cup \ r^{*} \tag{31}$$

where $r^{\star}$ is the reflexive and transitive closure of r. This proof is done in a similar manner as that of (30), keeping in mind that $r^{\star}$ is defined as follows:

$$r^{\star} \ \widehat{=} \ \mathsf{fix}(\lambda s \cdot \mathrm{id} \cup (r\,;s))$$

3.3 Modalities

In this section, we show how the two main basic modality operators, namely $\Box$ and $\Diamond$, can be given a formal set theoretic definition. We assume that we have an infinite process formalized by means of a total relation r (without pre-condition). To this forward relation there corresponds a backward set transformer F.

Always. Given a predicate $P(x)$, the predicate $\Box P(x)$ holds when $P(x)$ always holds as we move following the relation r. The set, $\{x \mid \Box P(x)\}$, is the set where the iterate
$(\{x \mid P(x)\} \Longrightarrow F)^{\triangledown}$ runs for ever, formally (according to (24)):

$$\{x \mid \Box P(x)\} \ = \ \overline{\mathsf{fix}(\{x \mid P(x)\} \Longrightarrow F)} \tag{32}$$

Eventually. Given a predicate $P(x)$, the predicate $\Diamond P(x)$ holds when $P(x)$ will hold eventually. In fact, $\Diamond P(x)$ can be defined as follows in terms of the $\Box$ operator

$$\Diamond P(x) \;\widehat{=}\; \neg \Box \neg P(x)$$

According to (32), we obtain the following:

$$\{x \mid \Diamond P(x)\} \;=\; \mathsf{fix}(\overline{\{x \mid P(x)\}} \Longrightarrow F) \tag{33}$$

This is very intuitive: in order to be sure that $P(x)$ holds eventually, we must be in the set guaranteeing that the iterate $(\overline{\{x \mid P(x)\}} \Longrightarrow F)^{\nabla}$ does terminate

4 Hiding

4.1 Non-homogenous Computations

So far, we supposed that computations defined transitions from a set A to itself. In this section, we generalize this by supposing that computations define transitions from a set A to a set B that is different from A. This generalization is straightforward, we have:

$$\begin{array}{l} r \subseteq A \times B \\ \\ p \subseteq A \\ \\ F \in \mathbb{P}(B) \to \mathbb{P}(A) \end{array} \tag{34}$$

The translation from one approach to the other is not modified, we have:

$$\begin{array}{l} r \;=\; \{x \mapsto y \mid x \in \overline{F(\overline{\{y\}})}\} \\ \\ p \;=\; F(B) \\ \\ F(q) \;=\; \begin{cases} \overline{r^{-1}[\overline{q}]} & \text{if } q \neq B \\ \\ p & \text{if } q = B \end{cases} \end{array} \tag{35}$$

A special case is one where r is a total surjection from A to B and p is A:

$$\begin{array}{l} r \in A \twoheadrightarrow B \\ p = A \end{array} \tag{36}$$

We have then

$$F(q) \;=\; \begin{cases} r^{-1}[q] & \text{if } q \neq B \\ \\ A & \text{if } q = B \end{cases} \tag{37}$$

4.2 Projection

Given a computation $S1$ from $A1$ to $A1$ (with backward conjunctive set transformer $F1$ and forward model $(p1, r1)$), we can project it to a computation $S2$ from $A2$ to $A2$ (with backward set transformer $F2$ and forward model $(p2, r2)$) by means of a projection function from $A1$ to $A2$. The latter is a total surjection pr:

$$pr \in A1 \twoheadrightarrow A2 \tag{38}$$

According to (37), this function pr defines a set transformer PR with

$$PR(q) \;=\; \begin{cases} pr^{-1}[q] & \text{if } q \neq A2 \\ A & \text{if } q = A2 \end{cases} \tag{39}$$

The converse pr^{-1} of pr defines a set transformer RP. The set transformer $F2$ is defined as follows:

$$F2(q) = RP(F1(PR(q))) = \begin{cases} \overline{pr[\overline{F1(pr^{-1}[q])}]} & \text{if } q \neq A2 \\ \overline{pr[\overline{F1(A1)}]} & \text{if } q = A2 \end{cases} \tag{40}$$

One can prove that $F2$ is conjunctive if $F1$ is (proof left to the reader. Hint: $p^{-1}(q1 \cap q2) \;=\; p^{-1}(q1) \cap p^{-1}(q2)$ since p is functional) :

$$F2(q1 \cap q2) \;=\; F2(q1) \cap F2(q2) \tag{41}$$

One can also prove (proof left to the reader. Hint: remember that F is conjunctive) that the forward model is the following:

$$\begin{aligned} r2 &= pr^{-1} \,;\, r1 \,;\, pr \\ p2 &= \overline{pr[\overline{p1}]} \end{aligned} \tag{42}$$

5 Refinement

Refinement is the process by which we can replace a, possibly abstract, computation by another one that behaves like the former but is, nevertheless, more concrete. As usual, we shall consider two kinds of refinement: algorithmic and data refinement respectively. An algorithmic refinement (section 5.1) is one where the abstract and refined computations are working within the same state space. A data refinement (section 5.2) is one where the abstract and refined computations are not working within the same state space.

5.1 Algorithmic Refinement

Let $S1$ and $S2$ be two computations. We say that $S1$ is refined to $S2$, if using $S2$ instead of $S1$ does not make any difference. In other words, in using $S2$, and as far as the results are concerned, we cannot detect that the computation is made by $S2$ rather than by $S1$.

Backward Approach. Let $F1$ and $F2$ be two set transformers working with the same carrier set A. They correspond to two computations $S1$ and $S2$. $S1$ is said to be refined to $S2$ when $F1(q)$ is included in $F2(q)$ for each subset q of A.

$$S1 \sqsubseteq S2 \;\widehat{=}\; \forall q \cdot F1(q) \subseteq F2(q) \tag{43}$$

The explanation is simple: in order to be in the after set q after an execution of the computation $S1$, we must, by definition, start in the before set $F1(q)$. Now, if $F1(q) \subseteq F2(q)$ then using $S2$ instead of $S1$, and starting again in the before-set $F1(q)$ implies that we are also in the before set $F2(q)$, hence the execution of $S2$ shall end up in q, as if we were using $S1$. Thus using $S2$ instead of $S1$ is safe with regard to our goal of ending up in q. Finally, if this is the case for any subset q of A then we can *always* use $S2$ safely instead of $S1$. In other words, $S2$ refines $S1$.

Forward Approach. For defining the refinement in the forward model, we simply translate the previous definition (43) into the (r, p) forward model. We can prove the following:

$$S1 \sqsubseteq S2 \;\widehat{=}\; p1 \subseteq p2 \;\wedge\; r2 \subseteq r1 \tag{44}$$

Properties of Algorithmic Refinement. We define these properties for the backward model only. They can easily be transfered to the forward model. The first property says that refining a part implies refining the all. It has to be proved for all combinators introduced in Section 2.4. All corresponding easy proofs are left to the reader. We only give here the proof for the iteration combinator:

$$S1 \sqsubseteq S2 \;\Rightarrow\; S1^{\triangledown} \sqsubseteq S2^{\triangledown} \tag{45}$$

Let $F1$ and $F2$ be the set transformers of $S1$ and $S2$. We have to prove:

$$\forall q \cdot F1(q) \subseteq F2(q) \;\Rightarrow\; (\forall q \cdot \mathsf{fix}(q \,|\, F1) \subseteq \mathsf{fix}(q \,|\, F2))$$

Proof

$\forall q \cdot F1(q) \subseteq F2(q)$ HYP1

$\mathsf{fix}(q \,|\, F1) \subseteq \mathsf{fix}(q \,|\, F2)$

$\Leftarrow$ According to Theorem 1 of **Appendix 1**

$(q \,|\, F1)(\mathsf{fix}(q \,|\, F2)) \subseteq \mathsf{fix}(q \,|\, F2)$

$\Leftrightarrow$ According to Theorem 2 of **Appendix 1**

$q \cap F1(\mathsf{fix}(q \,|\, F2)) \subseteq q \cap F2(\mathsf{fix}(q \,|\, F2))$

$\Leftarrow$

$F1(\mathsf{fix}(q \,|\, F2)) \subseteq F2(\mathsf{fix}(q \,|\, F2))$

$\Leftrightarrow$ According to HYP1

TRUE

End of Proof

Here are obvious properties of algorithmic refinement. Transitivity:

$$S1 \sqsubseteq S2 \ \wedge \ S2 \sqsubseteq S3 \ \Rightarrow \ S1 \sqsubseteq S3 \tag{46}$$

Extreme refinements:

$$\mathsf{abort} \ \sqsubseteq \ S \ \sqsubseteq \ \mathsf{magic} \tag{47}$$

Structure of refinement:

$$S1 \sqcap S2 \ \sqsubseteq \ \begin{matrix} S1 \\ S2 \end{matrix} \ \sqsubseteq \ S1 \sqcup S2 \tag{48}$$

5.2 Data Refinement

Informal Definition. Let $S1$ be a computation working with the set $A1$. Data-refinement may occur when we replace the computation $S1$ by a computation $S2$ working now with the set $A2$ different from $A1$. In the case of algorithmic refinement of Section 5.1 we could compare $S1$ and $S2$ because they worked with the same space set A. This is not any more possible in the present case. However, we suppose that the set $A1$ can be *projected* into a set $A3$ by means of a surjective function pr as explained in Section 4. Likewise, we suppose that the set $A2$ can also be projected into the *same set* $A3$ by means of a surjective function qr. The set $A3$ is said to be the one that can be *observed* from outside for both $S1$ and $S2$. The idea is then to compare the behaviors of $S1$ and $S2$ on their common observable set space $A3$.

Formal Definitions. Let $PR, F1, RP, QR, F2, RQ$ be set transformers corresponding to the relations and computations $pr, S1, pr^{-1}, qr, S2, qr^{-1}$ respectively. They are typed as follows:

$$PR \in \mathbb{P}(A3) \rightarrow \mathbb{P}(A1) \qquad F1 \in \mathbb{P}(A1) \rightarrow \mathbb{P}(A1) \qquad RP \in \mathbb{P}(A1) \rightarrow \mathbb{P}(A3)$$

$$QR \in \mathbb{P}(A3) \rightarrow \mathbb{P}(A2) \qquad F2 \in \mathbb{P}(A2) \rightarrow \mathbb{P}(A2) \qquad RQ \in \mathbb{P}(A2) \rightarrow \mathbb{P}(A3)$$

According to (43), we have thus the following for the backward model of the date-refinement operator $\sqsubseteq_d$:

$$S1 \sqsubseteq_d S2 \ \ \widehat{=} \ \ \forall q \cdot RP(F1(PR(q))) \Rightarrow RQ(F2(QR(q))) \tag{49}$$

Sufficient Conditions for Data-refinement. As condition (49) is not very easy to use in practice, we now introduce some *sufficient conditions* implying (49). For this, we introduce a relation w from $A2$ to $A1$ whose domain is exactly $A2$:

$$w \subseteq A2 \times A1 \qquad \mathrm{dom}(w) = A2 \tag{50}$$

This relation w must be *compatible* with projection functions pr and qr. In other words, w must link points y and x that are projected to the same point on $A3$, formally:

$$\forall\, x, y \cdot x \mapsto y \in w^{-1} \;\Rightarrow\; pr(x) = qr(y) \tag{51}$$

We prove now that it is equivalent to the following

$$w^{-1}; qr \subseteq pr \tag{52}$$

Let W be the set transformer corresponding to the relation w. It is typed as follows:

$$W \in \mathbb{P}(A1) \to \mathbb{P}(A2)$$

As a consequence, (52) can be put under the following form:

$$\forall q \cdot RP(q) \;\Rightarrow\; RQ(W(q)) \tag{53}$$

We thus prove the following:

$$(\forall\, x, y \cdot x \mapsto y \in w^{-1} \;\Rightarrow\; pr(x) = qr(y)) \;\Leftrightarrow\; w^{-1}; qr \subseteq pr \tag{54}$$

Proof

$$\begin{array}{ll}
 & \forall\, x, y \cdot x \mapsto y \in w^{-1} \;\Rightarrow\; pr(x) = qr(y) \\
\Leftrightarrow & \\
 & \forall\, x, y, z \cdot z = qr(y) \;\wedge\; x \mapsto y \in w^{-1} \;\Rightarrow\; z = pr(x) \\
\Leftrightarrow & \\
 & \forall\, x, z \cdot (\, \exists y \cdot z = qr(y) \;\wedge\; x \mapsto y \in w^{-1}\,) \;\Rightarrow\; z = pr(x) \\
\Leftrightarrow & \\
 & \forall\, x, z \cdot (\, \exists y \cdot x \mapsto y \in w^{-1} \;\wedge\; y \mapsto z \in qr\,) \;\Rightarrow\; z = pr(x) \\
\Leftrightarrow & \\
 & \forall\, x, z \cdot x \mapsto z \in (w^{-1}\,;qr) \;\Rightarrow\; x \mapsto z \in pr \\
\Leftrightarrow & \\
 & w^{-1}\,;qr \;\subseteq\; pr
\end{array}$$

End of Proof

From (52), we can deduce the following

$$qr \;\subseteq\; w\,;pr \tag{55}$$

This can be put under the following form:

$$\forall q \cdot W(PR(q)) \Rightarrow QR(q) \tag{56}$$

Proof

$$\begin{array}{ll}
 & w^{-1}\,;qr \;\subseteq\; pr \\
\Rightarrow & \\
 & w\,;w^{-1}\,;qr \;\subseteq\; w\,;pr \\
\Rightarrow & \qquad\qquad \mathsf{id} \subseteq w\,;w^{-1} \quad \text{since} \quad \mathrm{dom}(w) = A2 \quad (50) \\
 & qr \;\subseteq\; w\,;pr
\end{array}$$

End of Proof
Finally, we propose the following:

$$\forall q \cdot W(F1(q)) \Rightarrow F2(W(q)) \tag{57}$$

We have then to prove that the newly introduced conditions (53), (56), and (57) are sufficient to imply data-refinement as expressed in (49):

$$\forall q \cdot RP(F1(PR(q))) \Rightarrow RQ(F2(QR(q))) \tag{58}$$

Proof

$$\begin{array}{ll} & RP(F1(PR(q))) \\ \Rightarrow & \text{According to (53)} \\ & RQ(W(F1(PR(q)))) \\ \Rightarrow & \text{According to (57)} \\ & RQ(F2(W(PR(q)))) \\ \Rightarrow & \text{According to (56)} \\ & RQ(F2(QR(q))) \end{array}$$

End of Proof
To summarize at this point, the definition of w in (50), together with constraints (53) and (56), allows one to prove that the condition (57) is *sufficient* to obtain data-refinement as expressed by (49). Note that the condition (57) is independent from the projection functions pr and qr. The proofs have been done for the backward model: it can be done in a similar manner for the forward model (although a little more complicated).

6 Analysis

In this section, we study how the general framework developed in previous sections can be mapped into several computation paradigms. More precisely, in Section 2.4 we defined some elementary combinators, in this section we see how these combinators are used in different computation paradigms: some of them will not be used and some new combinators will be defined: we shall see that these new combinators are in fact not so new since they will be defined in terms of more elementary ones defined in Section 2.4.

For each computation paradigm we present a table explaining which combinators are used directly (column DIRECT) or indirectly (column INDIRECT). The definition of the indirect combinators are given at the end of each corresponding subsection. A third column (REASONING) indicates which model (forward or backward) is preferably taken to reason within the paradigm.

6.1 Sequential Programming Language

The model of formal sequential programming is clearly that indicated in the famous seminal paper of Hoare [13].

DIRECT	INDIRECT	REASONING
deterministic assignment sequence	conditional loop	forward (Hoare triple)

Conditional and loop combinators, can be defined in terms of more elementary combinators as follows[3]:

if b **then** $S1$ **else** $S2$ **end**	$(b \Longrightarrow S1) \sqcap (\neg b \Longrightarrow S2)$
while b **do** S **end**	$(b \Longrightarrow S)^{\triangledown}\ ;\ (\neg b \Longrightarrow \mathsf{skip})$

6.2 Dijkstra's Guarded Command Language

The guarded command language is presented in the famous paper [5] and book [6] written by E.W. Dijkstra. It is a very simple, yet quite powerful, programming notation.

DIRECT	INDIRECT	REASONING
deterministic assignment sequence	conditional (**if** ... **fi**) loop (**do** ... **od**)	backward

As previously in Section 6.1, here are definitions of the main constructs of Dijkstra's guarded command language[4]:

if $b1 \Longrightarrow S1 \,\square \ldots \square\, bn \Longrightarrow Sn$ **fi**	$b1 \Longrightarrow S1 \sqcap \ldots \sqcap bn \Longrightarrow Sn \sqcap$ $\neg\,(b1 \vee \ldots \vee bn) \Longrightarrow \mathsf{abort}$
do $b1 \Longrightarrow S1 \,\square \ldots \square\, bn \Longrightarrow Sn$ **od**	$(b1 \Longrightarrow S1 \sqcap \ldots \sqcap bn \Longrightarrow Sn)^{\triangledown}$; $\neg\,(b1 \vee \ldots \vee bn) \Longrightarrow \mathsf{skip}$

[3] In the case of the loop, the computation S is supposed to be non-guarded.
[4] In the case of the loop, the statements $S1, \ldots, Sn$ are supposed to be non-guarded.

6.3 The Specification Statement

Several people, among which Back [3], Morgan [15], [16], and Morris [18] proposed to introduce in programs some "specification" statements dealing directly with a post-condition and also possibly together with a pre-condition (in [15], [16]). The term "Specification Statement" was coined by Morgan. We extend the Dijkstra's guarded command language with such a statement:

Specification statement	$[b(x), a(x, x')]$	$b(x) \mid a(x, x')$

6.4 Action System

Action System is presented in [4]. An Action System model is defined by means of some *actions*, each of which being made up of a guard (a predicate) and a possible sequence of assignments. There exists also a special non-guarded initialising action. The operational description of an action system execution is as follows: after the initialisation, any action with a true guard is executed (and so on) until no guard action is true (in this case, the system stops: it is said to be deadlocked) or for ever if there always exists an action with a true guard. In other words, an action system is just made of a unique (always implicit) outermost do ... od loop as defined in Section 6.2.

DIRECT	INDIRECT	REASONING
deterministic assignment sequence guarding bounded choice		backward

6.5 Event-B

Event-B is presented in [2] and [20]. It has been strongly influenced by Action System. In fact, the operational "execution" of an Event-B model is similar to that of an Action System. What characterizes Event-B (and simplify things a lot) is the *removal* of pre-conditioning (as in Action System) and also that of sequencing. In other words all assignments done in an action (called an event in Event-B) are "executed" in parallel. Notice that this does not preclude to use

Event-B to develop sequential programs dealing with the classical conditional and loop: see chapter 15 of [2].

DIRECT	INDIRECT	REASONING
deterministic assignment non-deterministic assignment post-conditioning skipping guarding parallelism bounded choice unbounded choice		forward

Discussion. Quite often, a model described with Action System or with Event-B never stops. In this case there is no point in defining any *result* as well as a notion of termination. However, interesting outcomes requiring proofs are given by the modalities described in Section 3.3, namely invariance and reachability.

7 Conclusion

This paper presented some set-theoretical models of computations. Two main equivalent approaches were proposed in section 2: forward and backward. A special emphasis was put on the notion of finite or infinite iterations in section 3: for this an abstract notion of iteration was proposed and developed. Refinement was reminded and put into this set-theoretical setting in section 5. Finally, an analysis of various computation paradigms were reviewed in section 6.

More developments could have been proposed but was not possible due to the limitation of size for such a paper: more proofs (notice again that all important mentioned proofs were mechanically checked with the prover of the Rodin Platform [20]), more appendices could have been written in order to ease the reading and make the paper self-contained, more results concerning infinite iterations and modalities, more explanations on the various combinators of section 2.4, and so on.

References

1. Abrial, J.-R.: The B-book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)

3. Back, R.: On the Correctness of Refinement Steps in Program Development. Technical Report University of Helsinki (1978)
4. Back, R., Kurki-Suonio, R.: Distributed Cooperation with Action Systems. ACM Transaction on Programming Languages and Systems 10(4), 513–554 (1988)
5. Dijkstra, E.: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. CACM (1975)
6. Dijkstra, E.: A Discipline of Programming. Prentice-Hall (1976)
7. Gardiner, P., Morgan, C.: Data Refinement of Predicate Transformer. Theoretical Computer Science (1991)
8. He, J., Hoare, T., Sanders, J.: Data Refinement Refined. In: Robinet, B., Wilhelm, R. (eds.) ESOP 1986. LNCS, vol. 213, pp. 187–196. Springer, Heidelberg (1986)
9. Hehner, E.: A Practical Theory of Programming. Springer (1993)
10. Hesselink, W.: Programs, Recursion, and Unbounded Choice. Cambridge University Press (1992)
11. Hoare, T.: Programs are Predicates. Mathematical Logic and Programming Languages. Prentice-Hall (1985)
12. Hoare, T., He, J.: Unifying Theories of Programming. Prentice-Hall (1998)
13. Hoare, T.: An Axiomatic Basis for Computer Programming. CACM (1969)
14. Jones, C.: Program Specification and Validation in VDM. Logic of Programming and Calculi of Discrete Design. Springer (1987)
15. Morgan, C.: The Specification Statement. ACM Transactions on Programming Languages and Systems (1988)
16. Morgan, C.: Programming from Specification. Prentice-Hall (1990)
17. Morgan, C.: Of wp and CSP. In: Beauty is our Business. Springer (1990)
18. Morris, J.: A Theoretical Basis for Stepwise Refinement and the Programming Calculus. Science of Computer Programming (1987)
19. Nelson, G.: A Generalization of Dijkstra's Calculus. ACM TOPLAS (1989)
20. Rodin: Event-B and the Rodin Platform, `http://event-b.org`

Appendix 1: Definition and Properties of Fixpoints

Definition. Given a set S and a set function H defined as follows:

$$H \in \mathbb{P}(S) \to \mathbb{P}(S)$$

We define $\mathsf{fix}(H)$ and $\mathsf{FIX}(H)$ to be the following:

$$\begin{array}{lcl} \mathsf{fix}(H) & \widehat{=} & \bigcap\{k \mid H(k) \subseteq k\} \\ \mathsf{FIX}(H) & \widehat{=} & \bigcup\{k \mid k \subseteq H(k)\} \end{array} \qquad \text{Definition 1}$$

We notice that the expression $\bigcap\{k \mid H(k) \subseteq k\}$ is well defined since the set $\{k \mid H(k) \subseteq k\}$ is not empty (it contains S).

Lower and Upper Bounds. The first properties of $\mathsf{fix}(H)$ and $\mathsf{FIX}(H)$ is that they are respectively lower and upper bounds, formally:

$$\begin{array}{l} \forall k \cdot H(k) \subseteq k \;\Rightarrow\; \mathsf{fix}(H) \subseteq k \\ \forall k \cdot k \subseteq H(k) \;\Rightarrow\; k \subseteq \mathsf{FIX}(H) \end{array} \qquad \text{Theorem 1}$$

Greatest Lower and Least Upper Bounds. The second properties of $\mathsf{fix}(H)$ and $\mathsf{FIX}(H)$ is that they are respectively greatest lower and least upper bounds, formally:

$$\forall l \cdot (\forall k \cdot H(k) \subseteq k \;\Rightarrow\; l \subseteq k) \;\Rightarrow\; l \subseteq \mathsf{fix}(H)$$
$$\forall l \cdot (\forall k \cdot k \subseteq H(k) \;\Rightarrow\; k \subseteq l) \;\Rightarrow\; \mathsf{FIX}(H) \subseteq l \qquad \text{Theorem 2}$$

The Fixpoint Theorems. The third property is the important Knaster-Tarski theorem. It suppose that the function H is monotonic, that is

$$\forall k1, k2 \cdot k1 \subseteq k2 \;\Rightarrow\; H(k1) \subseteq H(k2) \quad \text{Monotonicity}$$

Here are the theorems saying that $\mathsf{fix}(H)$ and $\mathsf{FIX}(H)$ are fixpoints of H:

$$\mathsf{fix}(H) = H(\mathsf{fix}(H))$$
$$\mathsf{FIX}(H) = H(\mathsf{FIX}(H)) \qquad \text{Theorem 3}$$

Least and Greatest Fixpoints. Assuming again the Monotonicity of H, the next property says that $\mathsf{fix}(H)$ and $\mathsf{FIX}(H)$ are respectively the least and the greatest fixpoints:

$$\forall k \cdot k = H(k) \;\Rightarrow\; \mathsf{fix}(H) \subseteq k \subseteq \mathsf{FIX}(H) \quad \text{Theorem 4}$$

As a consequence, we have:

$$\mathsf{fix}(H) \subseteq \mathsf{FIX}(H) \quad \text{Theorem 5}$$

Relationship Between the Least and Greatest Fixpoints. Finally, we relate $\mathsf{fix}(H)$ and $\mathsf{FIX}(H)$. For this, we define a function $\widetilde{\_}$[5] as follows:

$$\widetilde{\_} \in (\mathbb{P}(S) \to \mathbb{P}(S)) \;\to\; (\mathbb{P}(S) \to \mathbb{P}(S))$$

together with the following for all set k:

$$\widetilde{H}(k) \;\widehat{=}\; \overline{H(\overline{k})} \quad \text{Definition 2}$$

We have then the following results:

$$\overline{\mathsf{fix}(H)} \;=\; \mathsf{FIX}(\widetilde{H})$$
$$\overline{\mathsf{FIX}(H)} \;=\; \mathsf{fix}(\widetilde{H}) \qquad \text{Theorem 6}$$

An interesting property of the function $\widetilde{\_}$ together with guarding and pre-conditioning is the following:

$$\widetilde{g \,|\, F} \;=\; g \Longrightarrow \widetilde{F} \qquad \text{(Notice: } (p \Longrightarrow G)(k) \;\widehat{=}\; \overline{p} \cup G(k)\text{)}$$

Thus we have the following for all subset q:

$$(\widetilde{g \,|\, F})(q) = \overline{g} \cup \overline{F(\overline{q})} \quad \text{Theorem 7}$$

[5] This function is called the "conjugate" in [17].

Appendix 2: Definition of Well-Founded Relations

We are given a set S and a binary relation r built on S:

$$r \subseteq S \times S$$

The relation r is said to be *well-founded*, denoted by $\mathsf{wfd}(r)$, if all paths built on r from any point x of S are *finite*. A relation r that is not well-founded thus contains infinite paths built on r. A subset l of S contains infinite paths if for any point x in l there exist a point y, also in l, related to x by means of r, formally:

$$\forall x \cdot x \in l \Rightarrow (\exists y \cdot y \in l \wedge x \mapsto y \in r)$$

that is

$$l \subseteq r^{-1}[l]$$

Since the empty set trivially enjoys this property, we can define a well-founded relation as one where the only set l with this property is the empty set, hence the formal definition of $\mathsf{wfd}(r)$:

$$\mathsf{wfd}(r) \;\widehat{=}\; \forall l \cdot l \subseteq r^{-1}[l] \Rightarrow l = \varnothing \quad \text{Definition 3}$$

Model-Based Mutation Testing of Reactive Systems

From Semantics to Automated Test-Case Generation

Bernhard K. Aichernig

Institute for Software Technology
Graz University of Technology, Austria
aichernig@ist.tugraz.at

Abstract. In this paper we give an overview of our work on combining model-based testing and mutation testing. Model-based testing is a black-box testing technique that avoids the labour of manually writing hundreds of test cases, but instead advocates the capturing of the expected behaviour in a model of the system-under-test. The test cases are automatically generated from this model. The technique is receiving growing interest in the embedded-systems domain, where models are the rule rather than the exception.

Mutation testing is a technique for assessing and improving a test suite. A number of faulty versions of a program-under-test are produced by injecting bugs into its source code. These faulty programs are called mutants. A tester analyses if his test suite can "kill" all mutants. We say that a test kills a mutant if it is able to distinguish it from the original. The tester improves his test suite until all faulty mutants get killed.

In model-based mutation testing, we combine the central ideas of model-based testing and mutation testing: we inject bugs in a model and generate a test suite that will kill these bugs. In this paper, we discuss its scientific foundations and tools. The foundations include semantics and conformance relations; the supporting tools involve model checkers, constraint solvers and SMT solvers.

1 Introduction

Is testing able to show the absence of bugs? The most prominent negative answer was given by the late Edsger Dijkstra: "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence." [15]. Dijkstra was always motivating the need for formally verified software. Of course, in general Dijkstra is right, in the same way as Popper was right, when he stated that we can never verify that a theory is correct by a finite set of experiments. In principle, only refutation (falsification) is possible [20]. However, this should not lead to an over-pessimistic judgement rejecting testing completely. This would be futile, since testing is the only way of building trust in a *running* system embedded in a complex environment. Testing is needed to check our assumptions. With wrong assumptions, even formally verified software

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 23–36, 2013.

may fail. A famous example of such a rare and subtle software bug was found in the binary search algorithm implemented in the Java JDK 1.5 library in 2006 [13].

As mentioned, Sir Karl Popper proposed the process of falsification. The idea is to build up trust by trying to disprove a theory. Translated to computer-based systems, we form a theory by modelling the system that is under investigation. We call these models *test models*. By testing, we try to disprove that the constructed system conforms to the test model. The tests are guided by educated guesses of possible faults that have been made during the construction.

If these falsification attempts fail, we build up trust. More important, this trust is measurable since we know what kind of challenges the system survived, i.e. what kind of faults are absent. In this paper we present our work on *model-based mutation testing* that follows this fault-oriented strategy. The main advantage of this testing technique is that it can guarantee the absence of specific faults.

Our goal is to generate a small set of test cases that cover these anticipated faults. This is in contrast to more traditional model-based testing approaches that often aim for structural model coverage, like e.g., state coverage or transition coverage.

The remainder of this paper is structured as follows. Next, in Section 2 we introduce mutation testing. Then, in Section 3 we explain the process of model-based mutation testing. In Section 4 we develop its general theory. In Section 5 the general theory is instantiated for transformational systems. In Section 6 we show how to handle reactive systems. Finally, in Section 7 we draw our conclusions.

2 Mutation Testing

Mutation testing is a way of assessing and improving a test suite by checking if its test cases can detect a number of injected faults in a program. The faults are introduced by syntactically changing the source code following patterns of typical programming errors. These deviations in the code are called mutations. The resulting faulty versions of the program are called mutants. Usually, each mutant includes only one mutation. Examples of typical mutations include renaming of variables, replacing operators, e.g., an assignment for an equivalence operator, and slightly changing Boolean and arithmetic expressions. Note that we only consider mutations that are syntactically correct. The number and kind of mutations depend on the programming language and are defined as so-called *mutation operators*.

A mutation operator is a rewrite rule that defines how certain terms in the programming language are replaced by mutations. For every occurrence of the term the mutation operator rewrites the original program into a new mutant. After a set of mutants has been generated, the test cases are run on the original and on each mutant. If a test case can distinguish a mutant from the original program, i.e. a test case passes the original, but fails on a mutant, we say that

```
1  object triangle {
2
3   def tritype(a : Int, b : Int, c: Int) = (a,b,c) match {
4    case _ if (a <= c-b) => "no triangle"
5    case _ if (a <= b-c) => "no triangle"
6    case _ if (b <= a-c) => "no triangle"
7    case _ if (a == b && b == c) => "equilateral"
8    case _ if (a == b) => "isosceles"
9    case _ if (b == c) => "isosceles"
10   case _ if (a == c) => "isosceles"
11   case _  => "scalene"
12   }
13 }
```

Fig. 1. Scala function returning the type of a triangle

this test case kills a mutant. The goal is to develop a test suite that kills all mutants.

Mutation testing can also be lifted to a test case generation technique. The aim is to automatically search for test cases that kill the mutants, i.e. the faults. However, this is still research as was recently pointed out in a survey on mutation testing: "There is a pressing need to address the, currently unresolved, problem of test case generation." [17]. It is the objective of our research to solve this problem.

Example 1 (Mutation of Programs). Consider the Scala program in Figure 1. The function `tritype` takes three lengths of a triangle and returns the resulting type of triangle. An example mutation operator could rewrite every equality into a greater-equal operator (== ⇒ >=) . This would produce five mutants, each containing exactly one mutation. For example, in the first mutant (Mutant 1), Line 7 would be replaced by `case _ if (a >= b && b == c) => "equilateral"`. □

Mutation can also be applied on the modelling level, as the following example illustrates.

Example 2 (Mutation of Models). Consider the UML diagram of a car alarm system in Figure 2. From the initial state *OpenAndUnlocked* one can traverse to *ClosedAndLocked* by closing all doors and locking the car. Actions of closing, opening, locking, and unlocking are modelled by corresponding signals *Close*, *Open*, *Lock*, and *Unlock*. The alarm system is armed after 20 seconds in *ClosedAndLocked*. Upon entry of the *Armed* state, the model calls the method *AlarmArmed.SetOn*. Upon leaving the state, which can be done by either unlocking the car or opening a door, *AlarmArmed.SetOff* is called. Similarly, when entering the *Alarm* state, the optical and acoustic alarms are enabled. When leaving the alarm state, either via a timeout or via unlocking the car, both acoustic and optical alarm is turned off. When leaving the alarm state after a timeout the system returns to an armed state only in case it receives a close signal. Turning off the acoustic alarm after 30 seconds is reflected in the time-triggered transition leading to the *Flash* sub-state of the *Alarm* state.

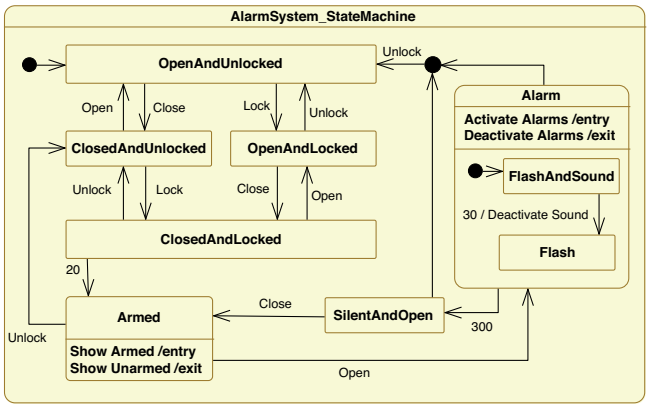


Fig. 2. State machine model of a car alarm system in UML

Let us consider a mutation operator for state machines that turns every transition into a reflexive transition ($\longrightarrow \Rightarrow \circlearrowright$). This operator produces 17 mutants of the car alarm system's UML diagram, one for each transition. For example, applying this mutation operator to the Lock-transition at the state *OpenAndUnlocked* results in a faulty behaviour staying in the state after a Lock-event. □

Our project partner, the Austrian Institute of Technology (AIT), has developed a large set of mutation operators for UML state machines, including removing trigger events on transitions, mutating transition signal events, mutating transition time trigger events, mutating transition OCL expressions, mutating transition effects, mutating transition guards, and removing entry and exit actions in states. We have recently studied the effectiveness of this mutation operators for different modelling styles [21]. This study shows that the number of generated mutants per mutation operator heavily depends on the style of the UML models.

After generating the mutants, we try to *kill* them. A test case kills a mutant, if its execution on the mutant shows a different behaviour than on the original. We say that a mutant survives a test case if it is not killed.

Example 3. Let us consider a set of test cases for the triangle example of Figure 1: `tritype(0,1,1)`, `tritype(1,0,1)`, `tritype(1,1,0)`, `tritype(1,1,1)`, `tritype(2,3,3)`, `tritype(3,2,3)`, `tritype(3,3,2)`, `tritype(2,3,4)`. These test cases cover all states, all branches, all paths of the program. They even satisfy the MC/DC coverage criterion and yet our mutant of Example 1 survives this test suite. For killing the mutant, we need an isosceles test case with $a > b$, e.g., `tritype(3,2,2)`. This test case kills the mutant by returning equilateral instead of isosceles. □

It is our goal to generate such test cases. This is not only possible for programs, but also for models.

Example 4. Let us return to the transition-mutation discussed in Example 2. This mutant may survive function coverage, state coverage and even transition

coverage, because the fault of staying in the state is only observable after waiting for 20 seconds and checking if the alarm system has been armed. Hence, a test sequence `Lock(); Close(); Wait(20)` is needed to kill this mutant. The expected behaviour of this test case is that the red flashing light indicating the arming will be switched on. In contrast, the mutant will show *quiescence*, i.e. the absence of any observation. □

In recent years, mutation testing has received a growing interest in academia [17]. Today, it is most frequently used as a technique to analyse the quality of a given test suite. The quality is measured in terms of the *mutation score* which is the ratio of killed mutants to the total number of mutants — the higher the mutation score, the better the test suite. Test suites can also be minimised, by reducing the number of test cases while keeping the mutation score. Obviously, the aim is to have a test suite with a maximal mutation score.

Our research aims at automatically generating the test cases that maximise the mutation score. Hence, rather than analysing a given test suite, we are interested in its synthesis. Our idea is to use and develop model checkers that analyse the equivalence between the original and a mutant. These tools produce a counter-example to equivalence which can be turned into a test case. However, there are some challenges.

Challenges. Unfortunately, achieving a mutation score of 1 (100%) is often impossible. The problem is that some mutants show an equivalent behaviour and, therefore, cannot be killed by any test case. The reason is that some syntactic changes do not have an effect on the semantics, e.g. mutations in code fragments that are never executed (dead code). Hence, these *equivalent mutants* have to be identified in order to normalise the mutation score[1]. For model checkers this means that in case of equivalence, the full state-space has to be explored, which may lead to the well-known state-space explosion. In general, equivalence checking is undecidable and NP-complete for bounded models. Therefore, we apply the technique to abstract models of the system-under-test (SUT). This leads to a combination of model-based testing and mutation testing, which we call *model-based mutation testing*.

3 Model-Based Mutation Testing

Figure 3 summarises the process of model-based mutation testing. Like in classical model-based testing, the user creates a test model out of the given requirements. A test case generator then analyses the model and generates an abstract test case (or a test suite). This test case is on the same abstraction level as the test model and includes expected outputs. A test driver maps the abstract test case to the concrete test interface of the SUT and executes the test case. The test driver compares the expected outputs with the actual outputs of the SUT and issues a verdict (pass or fail).

[1] Therefore, originally, the mutation score is defined as the ratio of killed mutants to the total number of *non-equivalent* mutants.

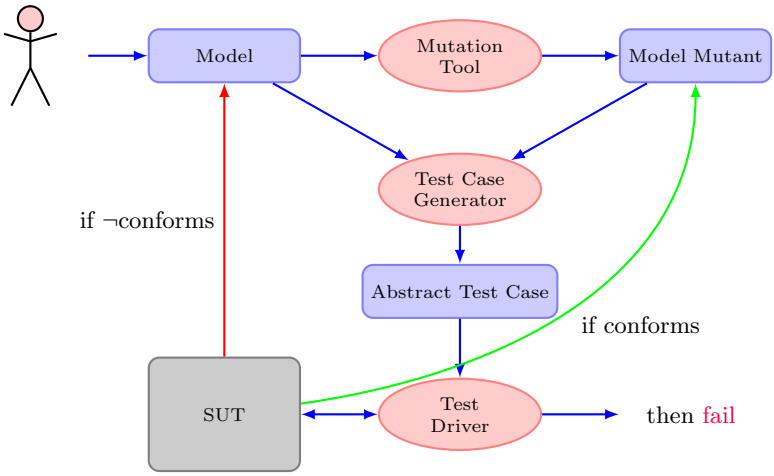


Fig. 3. Model-Based Mutation Testing

If the SUT conforms to the model, i.e. the SUT implements the model correctly, the verdict will always be *pass* (assuming that the tool chain generates sound test cases). In case of non-conformance (¬ conforms), i.e. a bug exists, we may issue a *fail* verdict. However, due to the incompleteness of testing, we may miss the bug and issue a *pass* verdict. Dijkstra was referring to these incompleteness of testing when he pointed out that testing cannot show the absence of bugs. However, in model-based mutation testing, we can improve this situation considerably.

In model-based mutation testing, we mutate the models automatically and then generate an abstract test case that will cover this mutation. What this coverage means will be defined later, when we define the conformance relation. For now we want to point out an important difference to other testing techniques: if a bug exists and this bug is represented by the generated mutant, then the test case will find this bug. This important property is illustrated in Figure 3 by the two conformance arrows: if the SUT does not conform to the model, but conforms to the mutant, the execution of the generated test case will result in a *fail* verdict. Here we are assuming a deterministic implementation. For non-deterministic SUTs, we have to repeat the test cases a given number of times.

4 General Theory

In this section we present the general theory of our model-based mutation testing approach. The theory is general in the sense that it does not define what kind of conformance relation is used. It can be any suitable order-relation. In the next sections, we will instantiate the conformance relation for transformational and reactive systems. The first property follows directly from Figure 3.

Theorem 1. *Given a transitive conformance relation* $\sqsubseteq$*, then*

$(Model \not\sqsubseteq SUT) \wedge (Mutant \sqsubseteq SUT) \;\Rightarrow\; (Model \not\sqsubseteq Mutant)$

Proof. *Proof by contradiction: let us assume* $Model \sqsubseteq Mutant$*, then by transitivity it follows from* $Mutant \sqsubseteq SUT$ *that* $Model \sqsubseteq SUT$*. This is a contradiction to the assumption* $Model \not\sqsubseteq SUT$*, hence* $Model \not\sqsubseteq Mutant$*.* □

The theorem expresses the fact that if a SUT has a fault and this fault is captured in the mutant, then the mutant is non-conforming to the model, i.e. the mutant is non-equivalent. Our test case generation algorithm is looking for the cases of non-conformance in $Model \not\sqsubseteq Mutant$. These cases are then turned into test cases and executed on the SUT. Such a test case will detect, if the SUT is an implementation of its $Mutant$.

Next, we characterize the test cases we are looking for. In general, a test case can be interpreted as a partial specification (model). It defines the expected output for one input and the rest is undefined. In this sense, a test case is highly abstract, because every behaviour different to its input-output is underspecified. This view causes sometimes confusion since the syntax of a test case is very concrete, but its semantics as a specification is very abstract. Consequently, if a SUT (always) passes a test case, we have conformance between the test case and the SUT:

$$Test\ case \sqsubseteq SUT$$

If we generate a test case from a model, we have selected a partial behaviour such that the model conforms to this test case:

$$Test\ case \sqsubseteq Model$$

If the SUT conforms to this model, we can relate all three:

$$Test\ case \sqsubseteq Model \sqsubseteq SUT$$

We can now define fault detecting test cases:

Definition 1. *Given a model and a mutant, its fault-detecting test case is (1) generated from the model and (2) kills the mutant, i.e.*

$$Test\ case \sqsubseteq Model \wedge Test\ case \not\sqsubseteq Mutant$$

Such a test case only exists for non-equivalent mutants:

Theorem 2

$Model \not\sqsubseteq Mutant$ **iff** $\exists Test\ case : (Test\ case \sqsubseteq Model \wedge Test\ case \not\sqsubseteq Mutant)$

The theorem shows that the fault-detecting test case is the counter-example to conformance. We presented the specification-view on test cases first in the weakest-precondition semantics of the refinement calculus [1,2]. The definition of fault-detecting test cases and their existence was developed in our mutation testing theory formulated in the relational semantics of the Unifying Theory of Programming (UTP) [6]. Next, we instantiate the general theory for transformational systems.

5 Transformational Systems

Transformational systems transform inputs and a pre-state to some output and post-state, then they terminate. Hence, the model and mutant of a transformational system can be interpreted as predicates $Model(s, s')$ and $Mutant(s, s')$ describing their state transformations ($s \rightarrow s'$). For such relational models, conformance is defined via implication in the standard way [16]:

Definition 2 (Conformance as Implication)

$$Model \sqsubseteq Mutant =_{df} \forall s, s' : Mutant(s, s') \Rightarrow Model(s, s')$$

Here conformance between a mutant and a model means that all behaviour of the mutant is allowed by the model. Consequently, non-conformance is expressed via the existence of a behaviour of the mutant that is not allowed by the model:

Theorem 3

$$Model \not\sqsubseteq Mutant \;=\; \exists s, s' : Mutant(s, s') \wedge \neg Model(s, s')$$

Note that this is a constraint satisfaction problem. Hence, a constraint solver can be used to search for a pre-state (input) s leading to the fault.

Example 5. Contract languages, like e.g. the Java Modelling Language (JML), support the specification of the transition relation of a method. A contract of our triangle example would look very similar to the Scala code in Figure 1. Their predicative semantics would be equivalent. Let us consider the semantics of our triangle example and its mutant.

$$Mutant(a, b, c, res') \wedge \neg Model(a, b, c, res') =_{df}$$

$$(\ldots \neg(a \leq c - b \vee a \leq b - c \vee b \leq a - c) \wedge (a \geq b \wedge b = c \wedge res' = \text{equilateral}))$$

$$\wedge$$

$$\neg(\ldots \neg(a \leq c - b \vee a \leq b - c \vee b \leq a - c) \wedge (a = b \wedge b = c \wedge res' = \text{equilateral}))$$

The arrow indicates the difference in the semantics due to the mutation. Simplifying the formula results in the condition that all fault-detecting test cases must satisfy: $a > b \wedge b = c \wedge res' = \text{equilateral}$. A constraint solver would produce, e.g., the solution $a = 3, b = 2, c = 2, res' = \text{equilateral}$. This input together with the expected output of the original comprises the fault-detecting test case $a = 3, b = 2, c = 2, res' = \text{isosceles}$. □

We developed this theory for transformational systems together with He Jifeng [6]. The technique was implemented with different solvers for different specification languages, e.g. OCL [11], Spec# [18], and the Reo connector language [19].

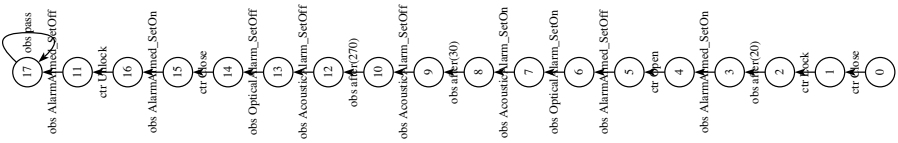


Fig. 4. An abstract test case for the car alarm system

6 Reactive Systems

Reactive systems continuously react to their environment and do not necessarily terminate. Common examples of such systems are controllers and servers. The points of observations from a tester's perspective are controllable (input) and observable (output) events. A test case for such systems is a sequence of controllable and observable events in the deterministic case. For non-deterministic systems, test cases have to branch over all possible observations. Such tree-like test cases are known as adaptive test cases.

The operational semantics of such systems is usually given in terms of *Labelled Transition Systems* (LTS) and the abstract test cases are LTS, too. Hence, in the deterministic case an abstract test case is a sequence of (input and output) labels.

Example 6. The car alarm system of Example 2 is a reactive system. Figure 4 shows a generated abstract test case for this system.

A prominent testing theory for this kind of semantics was developed by Tretmans [22]. Its conformance relation *ioco* is defined as follows.

Definition 3

$SUT\ \mathsf{ioco}\ Model =_{df} \forall \sigma \in \mathsf{traces}(Model) : \mathsf{out}(SUT\ \mathsf{after}\ \sigma) \subseteq \mathsf{out}(Model\ \mathsf{after}\ \sigma)$

Here after *denotes the set of reachable states after a trace* σ*, and* out *denotes the set of all observable events in a set of states. The observable events are all output events plus one additional quiescence event for indicating the absence of any output.*

This input-output conformance relation *ioco* supports non-deterministic models (see the subset relation) as well as partial models (only traces of the Model are tested). For input-complete models *ioco* is equivalent to trace-inclusion (language inclusion).

Example 7. The left-hand side of Figure 5 shows the LTS semantics of switching on both kind of alarms non-deterministically. Exclamation marks denote observable events, question marks controllable events. In this model either the flash, or the sound is switched on first. An implementer may decide for one of the two interleavings according to *ioco*. He might even add additional controllable events at any point, like the ?*unlock*-event in the LTS at the centre. However, the subset relation of output events has to be respected. Therefore, it is the *!soundOff* event in the mutant in the centre causing non-conformance. □

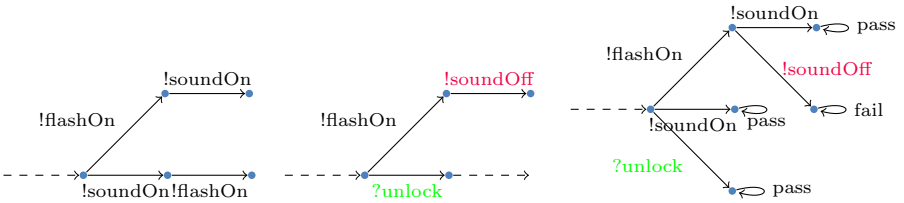


Fig. 5. Labelled transition systems of a part of a non-deterministic model of the car alarm system (left), a mutant (centre), and their synchronous product graph (right)

6.1 Explicit Conformance Checking

The conformance between a model and its mutant can be checked by building the synchronous product of their LTSs modulo *ioco*. The right-hand side of Figure 5 shows this product graph for our example. Product modulo *ioco* means that we limit the standard product construction if the mutant has either (1) an additional (unexpected) output event (here *!soundOff*), or (2) an additional input event (here ?*unlock*). In the first case, we have detected non-conformance and add a fail state after the unexpected event. Furthermore, we add all expected observables of the model. In the second case we stop the exploration, because we have reached an unspecified input behaviour.

Different strategies for extracting a test case from such a product graph exist. We can select a linear or adaptive test case, the shortest path or a random path to a fail-state, cover each fail-state or only one. Our experiments have shown that a combination of random and lazy shortest path strategies works well [3]. Lazy refers to the strategy of generating new test cases only, if the existing test cases do not kill a mutant.

We have applied this explicit conformance checking technique to generate test cases to several case studies, including testing an HTTP server using LOTOS models [5], SIP servers [23] using LOTOS models, controllers [3] using UML models, and most challenging, hybrid systems [14] using Action Systems extended with qualitative reasoning models [4].

Explicit checking works well with event-oriented systems, but we ran into scalability issues with parametrised events. Therefore, we have developed a second line of tools using symbolic conformance checkers.

6.2 Symbolic Conformance Checking

The idea is to use a similar approach as for transformational systems. Therefore, we have decided to use Back's Action Systems [12] as our input language. Action systems are a kind of guarded command language for modelling concurrent reactive systems. They are similar to Dijkstra's iterative statement.

Example 8. Figure 6 shows an Action System model of our car alarm system example. First, the model variables and their initial values are declared. Next,

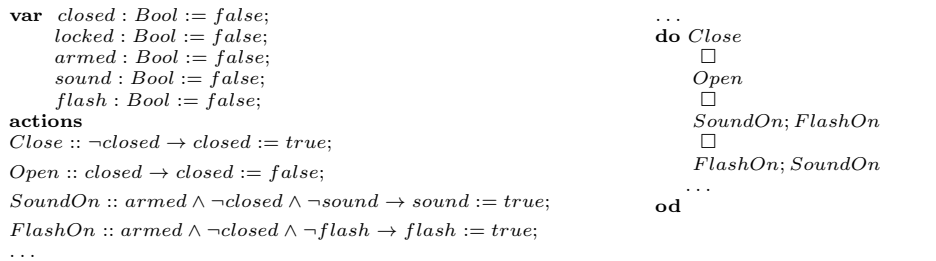


Fig. 6. Action System model of the car alarm system

the actions in the form of guarded commands are listed. Note that each action is labelled establishing the link to the LTS semantics. On the right-hand side is the protocol layer of actions which further restricts the possible order of actions. The standard composition operator for actions is non-deterministic choice ($A \Box B$), however also sequential ($A; B$) or prioritised compositions ($A // B$) are possible. The protocol layer establishes a loop that iterates while any action is enabled. Action Systems terminate if all actions are disabled. □

Originally, Actions Systems are defined in weakest-precondition semantics. However, for our purposes a relational semantics suffices. Therefore, we have given Action Systems a predicative semantics in the style of UTP as shown in Figure 7 [7].

The state-changes of actions are defined via predicates relating the pre-state of variables s and their post-state s'. Furthermore, the labels form a visible trace of events tr that is updated to tr' whenever an action runs through. Hence, a guarded action's transition relation is defined as the conjunction of its guard g, the body of the action B and the adding of the action label l to the previously observed trace. In case of parameters $\overline{x}$, these are added as local variables to the predicate. An assignment updates one variable x with the value of an expression e and leaves the rest unchanged. Sequential composition is standard: there must exist an intermediate state s_0 that can be reached from the first body predicate and from which the second body predicate can lead to its final state. Finally, non-deterministic choice is defined as disjunction. The semantics of the do-od block is as already mentioned: while actions are enabled in the current state, one of the enabled actions is chosen non-deterministically and executed. An action is enabled in a state if it can run through, i.e. if a post-state exists such that the semantic predicate can be satisfied.

This semantics is already close to a constraint satisfaction problem. However, the existential quantifiers need to be eliminated first, before we can negate the formula. For further details see [8].

With this predicative semantics conformance can be defined via implication, too. However, we also have to take the reachability via a trace of actions into account. For mutation testing, we are interested in non-conformance which can be defined as follows.

$$
\begin{array}{ll}
l :: g \to B & =_{df}\ g\ \wedge\ B\ \wedge\ tr' = tr \frown [l] \\
l(\overline{x}) :: g \to B & =_{df}\ \exists\ \overline{x} : g\ \wedge\ B\ \wedge\ tr' = tr \frown [l(\overline{x})] \\
x := e & =_{df}\ x' = e\ \wedge\ y' = y\ \wedge ... \wedge\ z' = z \\
g \to B & =_{df}\ g\ \wedge\ B \\
B(s, s'); B(s, s') & =_{df}\ \exists\ s_0 : B(s, s_0)\ \wedge\ B(s_0, s') \\
B_1 \,\square\, B_2 & =_{df}\ B_1\ \vee\ B_2
\end{array}
$$

Fig. 7. Predicative UTP semantics of Action System

Definition 4. *Given two Action Systems, a model and its mutant, then non-conformance is given iff*

$$\exists\ s, s', tr, tr' : reachable(s, tr)\ \wedge\ Mutant(s, s', tr, tr')\ \wedge\ \neg Model(s, s', tr, tr')$$

The predicate $reachable(s, tr)$ holds if a state s is reachable via trace tr from the initial state of the Action System.

This definition of non-conformance is much stronger than non-*ioco*, since any difference in the state s' causes non-conformance. Here, we also do not distinguish between input and output labels. However, it is very efficient to search for, since reachability can be done on the model only.

We have recently implemented two symbolic conformance checkers using this formula. One is implemented in Sicstus Prolog and uses Constraint Logic Programming. The other is implemented in Scala and uses the SMT solver Z3. In our first experiments both show similar performance [9].

For an *ioco* check of input-complete models, non-conformance amounts to a language-inclusion check:

$$
\begin{array}{c}
\exists\ s_1, s_1', s_2, s_2', tr, !a : reachable(Mutant, tr, s_1)\ \wedge\ reachable(Model, tr, s_2) \\
\wedge \\
Mutant(s1, s1', tr, tr \frown !a)\ \wedge\ \neg Model(s2, s2', tr, tr \frown !a)
\end{array}
$$

Here non-conformance is only due to a common trace leading to an output-label (output-action) in the mutant that is not allowed by the model. Note that this *ioco* formula holds for deterministic models only. In the non-deterministic case we have to check that none of the reachable states leads to an unexpected observation.

Most recently, we have implemented a similar formula in a test-case generator for timed automata (TA). The tool reads TA models and checks for *tioco*, a timed version of *ioco*. Here Scala and the Z3 solver are used [10].

7 Conclusions

We have shown how model-based testing and mutation testing can be combined into model-based mutation testing. We started with transformational systems

and then developed explicit and symbolic techniques for reactive systems. Several tools have been implemented that show that the approach is feasible.

The underlying test-case generation techniques are closely related to formal semantics. With a precise semantics we can define our notion of conformance. Non-conformance is the basis for our fault-models and test-case generation algorithms. Test cases are derived from counter-examples of a conformance check. With a predicative semantics such counter-examples may be found using constraint or SMT solvers.

The novelty in this research is the general theory and the test-case generators that can deal with non-deterministic models. For related work we refer to the recent survey on mutation testing, where also model-based mutation testing is covered [17].

The presented work shows that model-based mutation testing involves a variety of research directions and is far from being a closed case. As of today, no commercial tool has adopted this technique yet. Scalability is certainly an issue, but we firmly believe that advances are possible.

Acknowledgement. The recent research has received funding from the ARTEMIS Joint Undertaking under grant agreement N° 269335 and from the Austrian Research Promotion Agency (FFG) under grant agreement N° 829817 for the implementation of the project MBAT, Combined Model-based Analysis and Testing of Embedded Systems. The work was also funded by the Austrian Research Promotion Agency (FFG), program line "Trust in IT Systems", project number 829583, TRUst via Failed FALsification of Complex Dependable Systems Using Automated Test Case Generation through Model Mutation (TRUFAL).

References

1. Aichernig, B.K.: Test-case calculation through abstraction. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 571–589. Springer, Heidelberg (2001)
2. Aichernig, B.K.: Mutation Testing in the Refinement Calculus. Formal Aspects of Computing 15(2-3), 280–295 (2003)
3. Aichernig, B.K., Brandl, H., Jöbstl, E., Krenn, W.: Efficient mutation killers in action. In: IEEE Fourth International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, pp. 120–129. IEEE Computer Society (2011)
4. Aichernig, B.K., Brandl, H., Wotawa, F.: Conformance testing of hybrid systems with qualitative reasoning models. In: Finkbeiner, B., Gurevich, Y., Petrenko, A.K. (eds.) Proceedings of Fifth Workshop on Model Based Testing, MBT 2009, York, England, March 22. Electronic Notes in Theoretical Computer Science, vol. 253(2), pp. 53–69. Elsevier (October 2009)
5. Aichernig, B.K., Delgado, C.C.: From faults via test purposes to test cases: on the fault-based testing of concurrent systems. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 324–338. Springer, Heidelberg (2006)
6. Aichernig, B.K., He, J.: Mutation testing in UTP. Formal Aspects of Computing 21(1-2), 33–64 (2009)
7. Aichernig, B.K., Jöbstl, E.: Towards symbolic model-based mutation testing: Combining reachability and refinement checking. In: 7th Workshop on Model-Based Testing, MBT 2012. EPTCS, vol. 80, pp. 88–102 (2012)

8. Aichernig, B.K., Jöbstl, E.: Towards symbolic model-based mutation testing: Pitfalls in expressing semantics as constraints. In: Workshops Proc. of the 5th Int. Conf. on Software Testing, Verification and Validation, ICST 2012, pp. 752–757. IEEE Computer Society (2012)
9. Aichernig, B.K., Jöbstl, E., Kegele, M.: Incremental refinement checking for test case generation. In: Veanes, M., Viganò, L. (eds.) TAP 2013. LNCS, vol. 7942, pp. 1–19. Springer, Heidelberg (2013)
10. Aichernig, B.K., Lorber, F., Ničković, D.: Time for mutants: Model-based mutation testing with timed automata. In: Veanes, M., Viganò, L. (eds.) TAP 2013. LNCS, vol. 7942, pp. 20–38. Springer, Heidelberg (2013)
11. Aichernig, B.K., Salas, P.A.P.: Test case generation by OCL mutation and constraint solving. In: Cai, K.-Y., Ohnishi, A. (eds.) Fifth International Conference on Quality Software, QSIC 2005, Melbourne, Australia, September 19-21, pp. 64–71. IEEE Computer Society (2005)
12. Back, R.-J., Kurki-Suonio, R.: Decentralization of process nets with centralized control. In: Proceedings of the 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing, Montreal, Quebec, Canada, pp. 131–142. ACM (1983)
13. Bloch, J.: Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken. Google Research Blog (June 2006), `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html` (last visit May 17, 2013)
14. Brandl, H., Weiglhofer, M., Aichernig, B.K.: Automated conformance verification of hybrid systems. In: Wang, J., Chan, W.K., Kuo, F.-C. (eds.) Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, July 14-15, pp. 3–12. IEEE Computer Society (2010)
15. Dijkstra, E.W.: The humble programmer. Communications of the ACM 15(10), 859–866 (1972)
16. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall International (1998)
17. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5), 649–678 (2011)
18. Krenn, W., Aichernig, B.K.: Test case generation by contract mutation in Spec#. In: Finkbeiner, B., Gurevich, Y., Petrenko, A.K. (eds.) Proceedings of Fifth Workshop on Model Based Testing, MBT 2009, York, England, March 22. Electronic Notes in Theoretical Computer Science, vol. 253(2), pp. 71–86. Elsevier (October 2009)
19. Meng, S., Arbab, F., Aichernig, B.K., Astefanoaei, L., de Boer, F.S., Rutten, J.: Connectors as designs: Modeling, refinement and test case generation. Science of Computer Programming (2011) (in press, corrected proof)
20. Popper, K.: Logik der Forschung, 10th edn. Mohr Siebeck (2005)
21. Tiran, S.: On the effects of UML modeling styles in model-based mutation testing. Master's thesis, Institute for Software Technology, Graz University of Technology (2013)
22. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools 17(3), 103–120 (1996)
23. Weiglhofer, M., Aichernig, B., Wotawa, F.: Fault-based conformance testing in practice. International Journal of Software and Informatics 3(2-3), 375–411 (2009); Special double issue on Formal Methods of Program Development edited by Dines Bjoerner

Pliant Modalities in Hybrid Event-B

Richard Banach

School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
banach@cs.man.ac.uk

Abstract. Hybrid Event-B includes provision for continuously varying behaviour as well as the usual discrete changes of state in the context of Event-B. As well as being able to specify hybrid behaviour in the usual way, using differential equations or continuous assignments for the continuous parts of the behaviour, looser ways of specifying behaviour at higher levels of abstraction are extremely useful. Although the need for such looser specification can be met using the logic of the formalism, certain metaphors (or patterns) occur so often, and are so useful in practice, that it is valuable to introduce special machinery into the specification language, to allow these frequently occurring patterns to be readily referred to. This paper introduces such machinery into Hybrid Event-B.

1 Introduction

In today's ever-increasing interaction between digital devices and the physical world, formalisms are needed to express the more complex behaviours that this allows. Furthermore, these days, it is no longer sufficient to focus on isolated systems, as it is more and more the case that families of such systems are coupled together using communication networks, and can thus influence each others' working. So today *Cyber-Physical Systems* [19, 22, 1, 11] are the primary focus of attention. It is gratifying to note on the occasion of Jifeng He's festschrift, that the present author's own interest in such systems was sparked during cooperation with Prof. He's group at ECNU.

These new kinds of system throw up novel challenges in terms of design technique, as it is proving more and more difficult to ignore the continuous characteristics in their behaviours. Specifically, such technical challenges are being increasingly felt in the context of the B-Method [2, 3], where an increasing number of applications involve continuous behaviour of some sort in an essential way. Hybrid Event-B [10] has been introduced to bring new capabilities to traditional discrete Event-B [3], in order to address the challenges referred to.

Hybrid Event-B is a formalism that aims to provide a 'minimally invasive' extension of traditional discrete Event-B, capable of dealing with continuous behaviour as a first class citizen. As described in the next section, traditional discrete Event-B events serve as the 'mode events' that interleave the 'pliant events' of Hybrid Event-B. The latter express the continuously varying behaviour of a hybrid formalism including both kinds of event. In this manner, a rigorous link can be made between continuous and discrete update, as needed in these contemporary applications.

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 37–53, 2013.

Since Event-B may be seen as related to the Action Systems formalism of Back and co-workers [5, 6], so Hybrid Event-B may be seen as related to the Hybrid Action Systems that extend the Action Systems formalism [7, 9, 8]. However, there are some crucial differences. The most important of these is the fact that in Hybrid Action Systems, (pieces of) continuous behviours are packaged up into lambda abstractions (with time as the lambda variable), whereas discrete updates are handled as usual (i.e. by manipulating free variables). Although the approach is mathematically feasible, it introduces a discrepancy between the way that discrete and continuous updates are handled — and in fact, continuous updates are processed in discrete lumps, at occurrences of discrete updates, where the lambda abstractions are updated monolithically.[1]

From an applications perspective, it can be argued that this distinction is aesthetically jarring, but it also has technical repercussions. Most importantly, the mechanical processing of lambda abstractions, in practice, typically has much less power (in terms of the inferences that can be made automatically) than the mechanical processing of expressions predominantly featuring free variables. So automation will typically be less effective using such an approach. Hybrid Event-B treats the continuous behaviours via free variables, which does not in itself dilute the potential for mechanical processing.

Although Hybrid Event-B is a fully expressive formalism, based on the general theory of first order ordinary differential equations (ODEs) for the continuous part of the formalism and thus is capable of describing all the kinds of behaviour needed for arbitrary hybrid systems, it is nevertheless the case that in the continuous context there are metaphors that arise so commonly, that it is worth optimising the formalism to enable their even more convenient use. It is the aim of this paper to describe a representative selection of such optimisations, called here pliant modalities, and to show how their use can be integrated into the existing formalism of Hybrid Event-B.

The rest of the paper is as follows. Section 2 gives a brief description of Hybrid Event-B that is sufficient for the remainder. Section 3 discusses pliant modalities in general. The subsequent sections discuss specific pliant modalities in detail. Section 4 discusses the CONTINUOUS and similar modalities. Section 5 discusses the CONSTant modality. Section 6 discusses the PLiant ENVelope modality and related modalities. Sections 7 and 8 discuss monotonic modalities and convexity and concavity. Section 9 covers a small case study. Section 10 concludes.

2 Hybrid Event-B, a Sketch

In Fig. 1 we see a bare bones Hybrid Event-B machine, *HyEvBMch*. It starts with declarations of time and of a clock. In Hybrid Event-B time is a first class citizen in that all variables are functions of time, whether explicitly or implicitly. However time is special, being read-only, never being assigned, since time cannot be controlled by any human-designed engineering process. Clocks allow a bit more flexibility, since they are assumed to increase their value at the same rate that time does, but may be (re)set during mode events (see below).

[1] The discrete analogue of this would be to express every variable in conventional Event-B as a lambda function of the (normally implicit) indexing variable of a runtime trace of the system.

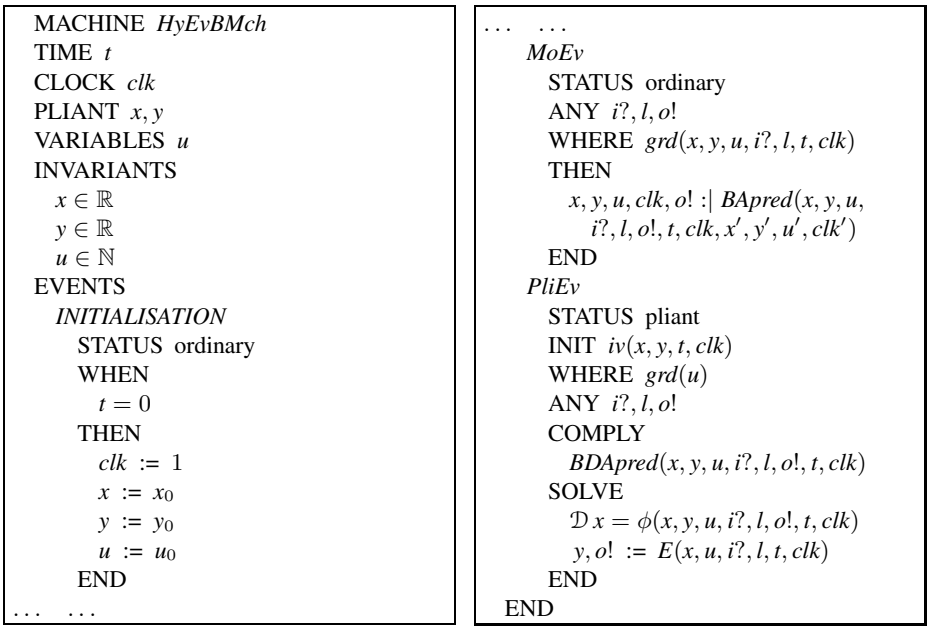


Fig. 1. A schematic Hybrid Event-B machine

Variables are of two kinds. There are mode variables (like u, declared as usual) which take their values in discrete sets and change their values via discontinuous assignment in mode events. There are also pliant variables (such as x, y), declared in the PLIANT clause, which take their values in topologically dense sets (normally $\mathbb{R}$) and which are allowed to change continuously, such change being specified via pliant events (see below).

Next are the invariants. These resemble invariants in discrete Event-B, in that the types of the variables are asserted to be the sets from which the variables' values *at any given moment of time* are drawn. More complex invariants are similarly predicates that are required to hold *at all moments of time* during a run.

Then we get to the events. The *INITIALISATION* has a guard that synchronises time with the start of any run, while all other variables are assigned their initial values in the usual way. As hinted above, in Hybrid Event-B, there are two kinds of event: mode events and pliant events.

Mode events are direct analogues of events in discrete Event-B. They can assign all machine variables (except time itself). In the schematic *MoEv* of Fig. 1, we see three parameters $i?, l, o!$, (an input, a local parameter, and an output respectively), and a guard *grd* which can depend on all the machine variables. We also see the generic after-value assignment specified by the before-after predicate *BApred*, which can specify how the after-values of all variables (except time, inputs and locals) are to be determined.

Pliant events are new. They specify the continuous evolution of the pliant variables over an interval of time. The schematic pliant event *PliEv* of Fig. 1 shows the structure. There are two guards: there is *iv*, for specifying enabling conditions on the pliant variables, clocks, and time; and there is *grd*, for specifying enabling conditions on the

mode variables. The separation between the two is motivated by considerations connected with refinement.

The body of a pliant event contains three parameters $i?, l, o!$, (once more an input, a local parameter, and an output respectively) which are functions of time, defined over the duration of the pliant event. The behviour of the event is defined by the COMPLY and SOLVE clauses. The SOLVE clause specifies behaviour fairly directly. For example the behaviour of pliant variable y and output $o!$ is given by a direct assignment to the (time dependent) value of the expression $E(\ldots)$. Alternatively, the behaviour of pliant variable x is given by the solution of the first order ordinary differential equation (ODE) $\mathcal{D}\,x = \phi(\ldots)$, where $\mathcal{D}$ indicates differentiation with respect to time. (In fact the sematics of the $y, o! = E$ case is given in terms of the ODE $\mathcal{D}\,y, \mathcal{D}\,o! = \mathcal{D}\,E$, so that x, y and $o!$ satisfy the same regularity properties.) The COMPLY clause can be used to express any additional constraints that are required to hold during the pliant event via its before-during-and-after predicate *BDApred*. Typically, constraints on the permitted range of values for the pliant variables, and similar restrictions, can be placed here.

The COMPLY clause has another purpose. When specifying at an abstract level, we do not necessarily want to be concerned with all the details of the dynamics — it is often sufficient to require some global constraints to hold which express the needed safety properties of the system. The COMPLY clauses of the machine's pliant events can house such constraints directly, leaving it to lower level refinements to add the necessary details of the dynamics. The kind of pliant modalities that are the main focus of this paper are frequently to be found in the COMPLY clauses of pliant events.

The semantics of a Hybrid Event-B machine is as follows. It consists of a set of *system traces*, each of which is a collection of functions of time, expressing the value of each machine variable over the duration of a system run. (In the case of *HyEvBMch*, in a given system trace, there would be functions for clk, x, y, u, each defined over the duration of the run.)

Time is modelled as an interval $\mathcal{T}$ of the reals. A run starts at some initial moment of time, t_0 say, and lasts either for a finite time, or indefinitely. The duration of the run $\mathcal{T}$, breaks up into a succession of left-closed right-open subintervals: $\mathcal{T} = [t_0 \ldots t_1), [t_1 \ldots t_2), [t_2 \ldots t_3), \ldots$. The idea is that mode events (with their discontinuous updates) take place at the isolated times corresponding to the common endpoints of these subintervals t_i, and in between, the mode variables are constant and the pliant events stipulate continuous change in the pliant variables.

Although pliant variables change continuously (except perhaps at the t_i), continuity alone still allows for a wide range of mathematically pathological behaviours. To eliminate these, we make the following restrictions which apply individually to every subinterval $[t_i \ldots t_{i+1})$:

I Zeno: there is a constant δ_{Zeno}, such that for all i needed, $t_{i+1} - t_i \geq \delta_{\mathsf{Zeno}}$.

II Limits: for every variable x, and for every time $t \in \mathcal{T}$, the left limit $\lim_{\delta\to 0} x(t-\delta)$ written $\overrightarrow{x(t)}$ and right limit $\lim_{\delta\to 0} x(t+\delta)$, written $\overleftarrow{x(t)}$ (with $\delta > 0$) exist, and for every t, $x(t) = \overleftarrow{x(t)}$. [N. B. At the endpoint(s) of $\mathcal{T}$, any missing limit is defined to equal its counterpart.]

III Differentiability: The behaviour of every pliant variable x in the interval $[t_i \ldots t_{i+1})$ is given by the solution of a well posed initial value problem $\mathcal{D}\,xs = \phi(xs \ldots)$ (where xs is a relevant tuple of pliant variables and $\mathcal{D}$ is the time derivative). "Well posed" means that $\phi(xs \ldots)$ has Lipschitz constants which are uniformly bounded over $[t_i \ldots t_{i+1})$ bounding its variation with respect to xs, and that $\phi(xs \ldots)$ is measurable in t.

Regarding the above, the Zeno condition is certainly a sensible restriction to demand of any acceptable system, but in general, its truth or falsehood can depend on the system's full reachability relation, and is thus very frequently undecidable.

The stipulation on limits, with the left limit value at a time t_i being not necessarily the same as the right limit at t_i, makes for an easy interpretation of mode events that happen at t_i. For such mode events, the before-values are interpreted as the left limit values, and the after-values are interpreted as the right limit values.

The differentiability condition guarantees that from a specific starting point, t_i say, there is a maximal right open interval, specified by t_{MAX} say, such that a solution to the ODE system exists in $[t_i \ldots t_{\mathrm{MAX}})$. Within this interval, we seek the earliest time t_{i+1} at which a mode event becomes enabled, and this time becomes the preemption point beyond which the solution to the ODE system is abandoned, and the next solution is sought after the completion of the mode event.

In this manner, assuming that the *INITIALISATION* event has achieved a suitable intial assignment to variables, a system run is *well formed*, and thus belongs to the semantics of the machine, provided that at runtime:

- Every enabled mode event is feasible, i.e. has an after-state, and on its completion enables a pliant event (but does not enable any mode event). (1)
- Every enabled pliant event is feasible, i.e. has a time-indexed family of after-states, and EITHER: (2)

  (i) During the run of the pliant event a mode event becomes enabled. It preempts the pliant event, defining its end. ORELSE
  (ii) During the run of the pliant event it becomes infeasible: finite termination. ORELSE
  (iii) The pliant event continues indefinitely: nontermination.

Thus in a well formed run mode events alternate with pliant events.[2] The last event (if there is one) is a pliant event (whose duration may be finite or infinite).

[2] Many formalisms for hybrid systems permit a succession of mode events to execute before the next pliant event runs (to use our terminology). We avoid this for a number of reasons. Firstly, it spoils the simple picture that at each time, each variable has a unique value, and the runtime semantics of a variable is a straightforward function of time. Secondly, it avoids having to define the final value of a succession of mode events via a fixpoint calculation. Thirdly, and perhaps most importantly, it maintains the discrete Event-B picture in which events are (implicilty) seen as taking place at isolated points of real time, insofar as Event-B behaviours are seen as relating to the real world. We regard the overturning of such unstated assumptions as particularly dangerous in an engineering context — c.f. the Mars Lander incident, in which the U.S. and European teams interpreted measurements according to different units, without ever thinking to check which units were actually intended.

We note that this framework is quite close to the modern formulation of hybrid systems. (See eg. [20, 14] for representative formulations, or the large literature in the *Hybrid Systems: Computation and Control* series of international conferences, and the further literature cited therein.)

In reality, there are a number of semantic issues that we have glossed over in the framework just sketched. We refer to [10] for a more detailed presentation. As well as these, in line with the normal B-Method approach, there is a full suite of proof obligations that statically guarantees conformance with the semantics (see [10] again). We do not have space to quote them all, but we overview the ones that are most relevant to the remainder of the paper. First we quote pliant event feasibility:

$$\begin{array}{l} I(u(\mathbb{t}_{\mathrm{L}})) \wedge iv_{PliEvA}(u(\mathbb{t}_{\mathrm{L}})) \wedge grd_{PliEvA}(u(\mathbb{t}_{\mathrm{L}})) \\ \quad \Rightarrow (\exists\, \mathbb{t}_{\mathrm{R}} > \mathbb{t}_{\mathrm{L}} \bullet [\, (\mathbb{t}_{\mathrm{R}} - \mathbb{t}_{\mathrm{L}} > \delta_{\mathsf{Zeno}PliEvA}) \wedge \,] \\ \qquad (\forall\, \mathbb{t}_{\mathrm{L}} < t < \mathbb{t}_{\mathrm{R}} \bullet (\exists\, u(t) \bullet BDApred_{PliEvA}(u(t), t) \wedge SOL_{PliEvA}(u(t), t)))) \end{array} \tag{3}$$

In (3), I is the machine invariant, *iv* and *grd* are guards, *PliEv* is the pliant event in question, u refers to all the machine's variables as needed, *SOL* is the solution to the SOLVE clause of *PliEv*, $\mathbb{t}_{\mathrm{L}}$ and $\mathbb{t}_{\mathrm{R}}$ define the endopints of the interval in which the solution holds, and $\delta_{\mathsf{Zeno}PliEv}$ is the relevant Zeno constant — the term containing it can be omitted if it is too difficult to establish Zeno-freeness statically. Note that both *BDApred* and *SOL* have to hold throughout the interval. If either fails to do so, it signals the end of the feasible interval for *PliEv*. Next we quote pliant event invariant preservation:

$$\begin{array}{l} I(u(\mathbb{t}_{\mathrm{L}})) \wedge iv_{PliEvA}(u(\mathbb{t}_{\mathrm{L}})) \wedge grd_{PliEvA}(u(\mathbb{t}_{\mathrm{L}})) \wedge \\ (\exists\, \mathbb{t}_{\mathrm{R}} > \mathbb{t}_{\mathrm{L}} \bullet (\forall\, \mathbb{t}_{\mathrm{L}} < t < \mathbb{t}_{\mathrm{R}} \bullet BDApred_{PliEvA}(u(t), t) \wedge SOL_{PliEvA}(u(t), t))) \\ \quad \Rightarrow (\forall\, \mathbb{t}_{\mathrm{L}} < t < \mathbb{t}_{\mathrm{R}} \bullet I(u(t))) \end{array} \tag{4}$$

The last PO we quote is the correctness PO for pliant event refinement:

$$\begin{array}{l} I(u(\mathbb{t}_{\mathrm{L}})) \wedge K(u(\mathbb{t}_{\mathrm{L}}), w(\mathbb{t}_{\mathrm{L}})) \wedge iv_{PliEvC}(w(\mathbb{t}_{\mathrm{L}})) \wedge grd_{PliEvC}(w(\mathbb{t}_{\mathrm{L}})) \Rightarrow \\ \big(\, \exists\, \mathbb{t}_{\mathrm{R}} > \mathbb{t}_{\mathrm{L}} \bullet (\forall\, \mathbb{t}_{\mathrm{L}} < t < \mathbb{t}_{\mathrm{R}} \bullet BDApred_{PliEvC}(w(t), t) \wedge SOL_{PliEvC}(w(t), t)) \\ \quad \Rightarrow (\forall\, \mathbb{t}_{\mathrm{L}} < t < \mathbb{t}_{\mathrm{R}} \bullet (\exists\, u(t) \bullet \\ \qquad BDApred_{PliEvA}(u(t), t) \wedge SOL_{PliEvA}(u(t), t) \wedge K(u(t), w(t))))\big) \end{array} \tag{5}$$

In (5), *PliEvA* and *PliEvC* are the abstract and concrete pliant events. Furthermore, the form of (5) implies that time progresses in the same way in the abstract and concrete systems. This is a consequence of the single outer level quantification over time $\forall\, \mathbb{t}_{\mathrm{L}} < t < \mathbb{t}_{\mathrm{R}}$, indicated by the heavy parentheses. The uniform parameterisation over time implies that (5) demands that the joint invariant $K(u(t), w(t))$ holds throughout the two pliant events.

3 Pliant Modalities and Requirements in Hybrid Event-B

The framework described above, when elaborated in full detail, is certainly expressive enough to subsume the range of problems tackled in the hybrid and cyber-physical systems domain. However it does so by reducing all aspects of system behaviour to their descriptions in a language that is essentially first order logic with real and integer arithmetic, real functions and set theory, supplemented by differential equations. Any such

language is, of course, rich enough to be highly undecidable. Despite this, many simple and intuitive properties of system behaviour nevertheless have descriptions in such a language that obscures their simplicity. This has two negative consequences. Firstly, it obscures readability and perspicuity for the user or designer, when straightforward ideas have to be written in a convoluted way. Secondly, it may make it relatively more difficult for a mechanised reasoning system to reach the most frequently intended consequences of these simple properties, when they are needed in verification.

The aim of this paper is to introduce a range of syntactic primitives for properties of real functions that occur commonly in hybrid applications, along with their definitions. Not only can this make system models more readable, but it can also be exploited by reasoning systems in order to optimise verification.

The primitives we introduce are in many ways really rather simple: constancy, boundedness, monotonicity, and so on. While easy to grasp, their defnitions in terms of base logical primitives are always more complicated than one feels they ought to be. As well as that though, and in stark contrast to the situation for purely discrete applications, there are not-so-obvious connotations with requirements that are worth exploring.

In a purely discrete application, when we write an oversimplified abstract model (as we are strongly encouraged to do at the outset of system development in Event-B), we are never in any doubt that what we define (as long as it is not patently ridiculous), is ultimately implementable. The basic reason for this is that the discrete data types we use in such early models are clearly implementable on the discrete hardware that we ultimately target them to. The case with continuously varying quantities is subtly different. Usually, we build continuous models to reflect the capabilities of real physical phenomena or real physical equipment. In such cases, the modelling flexibility that we have is severely curtailed, because the physical behaviour that we want to model is normally confined within quite tightly constrained parameters. Postulating a behaviour, no matter how desirable, is pointless unless we are confident about ultimate implementabiliy. In practice this usually means that we have to start modelling at a level considerably closer to the physical level than we might like. Only when we know that we are working within realistic parameters, can we permit ourselves to abstract from the details, to create a simplified model that deals just with the coarsest aspects of the dynamics.

Having pursued a strategy as just described, it would be reasonable at this point to question the purpose of the simplified model, given that a more precise description is already available. The response to that would be, that there could easily be properties of the system that are much more convenient to deal with when cast in in terms of the simplified model than they would be if cast in in terms of the more precise model. *In extremis*, some properties might only be tractable in the simplified model.

We infer that consideration of the continuous sphere can bring with it an inversion of the usual Event-B relationship between requirements and refinement. Usually in Event-B, once having incorporated a requirement into a system model, further models of the development are refinements of it. In the continuous case though, we have argued that a subsequent model might be an abstraction instead.

Technically, a pliant modality is a property of a pliant variable that is given a specific name. Such modalities may occur in two places in a Hybrid Event-B machine. One possibility is in the INVARIANTs of the machine. The properties defined by such

modalities must hold throughout any run of the machine. The other possibility is in the COMPLY clause of a pliant event. In this case, the relevant property is only required to hold during the execution of the pliant event, and indeed, if at some point during the execution of the pliant event, the modality contradicts other properties in the event's definition, it merely serves to define the limit of feasibility of the pliant event (i.e. PO (3) fails), signalling termination. We now discuss some modalities in more detail.

4 The CONTINUOUS Modality and Its Relatives

Our first, and simplest, modality is the continuous modality, which declares that a pliant variable's behaviour is (globally) continuous throughout the duration of any run of the system. Such a restriction is appropriate for a model of a physical variable which is not expected to undergo impulses during the dynamics, and would be written amongst the INVARIANTS. Since the semantics of Hybrid Event-B ensures that the behaviour of any pliant variable is absolutely continuous during any pliant event, the continuous modality just prevents the variable's value from being discontinuously reassigned by a mode event, a condition that is particularly easy to enforce statically. We write the continuous modality thus:

$$\mathsf{CONTINUOUS}(f) \;\equiv\; \ldots \tag{6}$$

where the ellipsis stands for one of a number of equivalent definitions of absolute continuity. (See e.g., [21] for details.) Related to the CONTINUOUS modality, and a little harder to enforce, are modalities that assert the derivative, or n-th derivative of f are globally continuous:

$$\mathsf{DIFFERENTIABLE}(f) \;\equiv\; \mathsf{CONTINUOUS}(\mathcal{D}f) \tag{7}$$

$$n\text{-}\mathsf{DIFFERENTIABLE}(f) \;\equiv\; \mathsf{CONTINUOUS}(\mathcal{D}^n f) \tag{8}$$

5 The CONST Modality

Our next modality is the constant modality, which declares that a pliant variable remains constant. At this point, the reader may well question the need for such a modality. Surely, if a variable is to be constant, aside from the possibility of declaring a constant instead, we could declare a mode variable, and declare it as CONTINUOUS. And there are other, similar possibilities. Pursuing such reasoning, the case we cannot cover by existing means is when we need to introduce a quantified variable, which we require to remain constant within the scope of the COMPLY clause of a pliant event, but whose value is not determinable statically. The definition of the constant modality for a variable f is thus:

$$\mathsf{CONST}(f) \;\equiv\; (\forall\, t \bullet \mathbb{t}_{\mathrm{L}} \leq t < \mathbb{t}_{\mathrm{R}} \Rightarrow f(\mathbb{t}_{\mathrm{L}}) = f(t)) \tag{9}$$

The form of the CONST modality that we show in (9) is the one that is appropriate for a COMPLY clause. This is characterised by the presence of $\mathbb{t}_{\mathrm{L}}$ and $\mathbb{t}_{\mathrm{R}}$ in (9). These

identifiers are free in (9) and bind to the intial and final times repectively of the duration of the pliant event at runtime. (The former fact follows from the assumption of the pliant event's guards as properties of $\mathbb{t}_{\mathrm{L}}$ in POs featuring $\mathbb{t}_{\mathrm{L}}$, and the latter from the assumption that $\mathbb{t}_{\mathrm{R}} > \mathbb{t}_{\mathrm{L}}$ in these POs (which is usually sufficient) — as is done in (3)-(5).) The form of (9) appropriate for an INVARIANT follows by removing the references to $\mathbb{t}_{\mathrm{R}}$ and replacing references to $\mathbb{t}_{\mathrm{L}}$ by references to the initial time t_0, leaving:

$$\mathsf{CONST}(f) \;\equiv\; (\forall\, t \bullet t_0 \leq t \Rightarrow f(t_0) = f(t)) \tag{10}$$

The CONST modality has an associated form (in which E is an expression) for cases when we can determine the value that f is to have during the relevant interval:

$$\mathsf{CONST}(f, E) \;\equiv\; (\forall\, t \bullet \mathbb{t}_{\mathrm{L}} \leq t < \mathbb{t}_{\mathrm{R}} \Rightarrow f(\mathbb{t}_{\mathrm{L}}) = f(t) = E(\mathbb{t}_{\mathrm{L}})) \tag{11}$$

In future, we will just show the COMPLY form of any modality, as the INVARIANT form follows readily, by simply removing references to $\mathbb{t}_{\mathrm{R}}$, and replacing values at $\mathbb{t}_{\mathrm{L}}$ with intial values where necessary.

6 The PLENV Modality and Its Relatives

The PLENV modality is at the opposite end of the expressivity spectrum to CONST. It constrains its argument f to remain within a PL*iant* ENV*elope* specified by a lower bound function and/or an upper bound function. It is easiest to build up from the simplest cases, so we start with dynamic lower and dynamic upper bounds alone:

$$\mathsf{PLENVL}(f, lb) \;\equiv\; (\forall\, t \bullet \mathbb{t}_{\mathrm{L}} \leq t < \mathbb{t}_{\mathrm{R}} \Rightarrow lb(t) \leq f(t)) \tag{12}$$

$$\mathsf{PLENVU}(f, ub) \;\equiv\; (\forall\, t \bullet \mathbb{t}_{\mathrm{L}} \leq t < \mathbb{t}_{\mathrm{R}} \Rightarrow f(t) \leq ub(t)) \tag{13}$$

The general dynamically bounded PLENV modality now follows:

$$\mathsf{PLENV}(f, lb, ub) \;\equiv\; \mathsf{PLENVL}(f, lb) \wedge \mathsf{PLENVU}(f, ub) \tag{14}$$

From these forms, several useful forms follow by restricting the various bounds involved to constants. We can easily define these by combining our existing definitions with the CONST modality, which is actually the most transparent way to do it.

The first two modalities just restrict PLENVL and PLENVU, giving conventional notions of lower and upper bounds:

$$\mathsf{LBND}(f, E) \;\equiv\; (\exists\, lb \bullet \mathsf{PLENVL}(f, lb) \wedge \mathsf{CONST}(lb, E)) \tag{15}$$

$$\mathsf{UBND}(f, E) \;\equiv\; (\exists\, ub \bullet \mathsf{PLENVU}(f, ub) \wedge \mathsf{CONST}(ub, E)) \tag{16}$$

Associated with these is the BND modality, combining the two of them:

$$\mathsf{BND}(f, E_L, E_U) \;\equiv\; \mathsf{LBND}(f, E_L) \wedge \mathsf{UBND}(f, E_U) \tag{17}$$

Finally we have versions of PLENV in which one bound but not the other is held constant:

$$\mathsf{PLENVLC}(f, E_L, ub) \;\equiv\; \mathsf{LBND}(f, E_L) \wedge \mathsf{PLENVU}(f, ub) \tag{18}$$

$$\mathsf{PLENVUC}(f, lb, E_U) \;\equiv\; \mathsf{PLENVL}(f, lb) \wedge \mathsf{UBND}(f, E) \tag{19}$$

The above pliant modalities give us a selection of primitives that can capture high level aspects of pliant variable behaviour in a concise and perspicuous way. Of course the aspects thus captured are not the only ones that can be exhibited by pliant variables, but they are typical of the relatively simple properties favoured in many engineering designs. We can thus expect them to have wide applicability, especially when they are combined with the possibility of making f a real-valued function of several variables (each depending on t). Moreover, we have to be sure that particular constants or bounds used are actually achievable, so design of these high level properties usually goes hand in hand with the lower level design that refines/implements it.

7 Monotonic Modalities

In this section we consider modalities connected with monotonicity. For functions of time delivering a single real value, the following are the evident possibilities.

$$\mathsf{MONINC}(f) \;\equiv\; (\forall\, t_1, t_2 \bullet \mathbb{t}_{\mathrm{L}} \leq t_1 \leq t_2 < \mathbb{t}_{\mathrm{R}} \Rightarrow f(t_1) \leq f(t_2)) \tag{20}$$

$$\mathsf{MONDEC}(f) \;\equiv\; (\forall\, t_1, t_2 \bullet \mathbb{t}_{\mathrm{L}} \leq t_1 \leq t_2 < \mathbb{t}_{\mathrm{R}} \Rightarrow f(t_1) \geq f(t_2)) \tag{21}$$

Compared with the modalities of Section 6, the above modalities can be used more freely at an abstract level, since they do not assert specific numerical measures for the rate of increase/decrease, making it relatively easier to postpone the determination of these rates during the development.

8 Convex and Concave Modalities

In the last section, the key properties depended on comparing the function under consideration at two points. Convexity and concavity depend on a comparison at three points, and are essentially concerned with capturing liberal notions of a function increasing or decreasing in an "accelerating" manner (bending upwards — convexity), or in a "decelerating" manner (bending downwards — concavity):

$$\begin{aligned}\mathsf{CVEX}(f) \;\equiv\; &(\forall\, t_1, t_2, \lambda \bullet \mathbb{t}_{\mathrm{L}} \leq t_1 \leq t_2 < \mathbb{t}_{\mathrm{R}} \wedge 0 < \lambda < 1 \Rightarrow \\ &f(\lambda\, t_1 + (1-\lambda) t_2) \leq \lambda f(t_1) + (1-\lambda) f(t_2))\end{aligned} \tag{22}$$

$$\begin{aligned}\mathsf{CCAVE}(f) \;\equiv\; &(\forall\, t_1, t_2, \lambda \bullet \mathbb{t}_{\mathrm{L}} \leq t_1 \leq t_2 < \mathbb{t}_{\mathrm{R}} \wedge 0 < \lambda < 1 \Rightarrow \\ &f(\lambda\, t_1 + (1-\lambda) t_2) \geq \lambda f(t_1) + (1-\lambda) f(t_2))\end{aligned} \tag{23}$$

These modalities evidently have similar levels of flexibility for abstract use as we saw in the previous section. Equally evidently, we can imagine more and more complex properties built along similar lines.

9 A Simple Case Study

In this section we give a simple case study, based on examples in [9, 7, 8, 17, 18, 4], and simplified for a briefer description. The case study is based on a nuclear power

station, where the reactor naturally tends to heat up despite heat being extracted for steam generation. To keep the temperature θ under control, a control rod, which absorbs neutrons, is inserted into the reactor, and the resulting lower neutron flux slows down the nuclear reaction, lowering the temperature. To prevent the reaction from stopping completely, the control rod is withdrawn after a period, and must cool down before it can be used again. In the works cited, more than one rod is used, leading to an interesting interplay between heating and cooling periods, and the recovery time of the rods.

For our purposes we simplify matters by having only one control rod, by assuming that the heating and cooling periods are equal, both lasting 3 time units, and that the 3 time units of the reactor heating period are sufficient for the control rod (which was extracted at the start of the reactor heating period) to itself have cooled down enough for rod reuse at the start of the next reactor cooling period.

In Fig. 2 we show a number of Hybrid Event-B machines that address this scenario. We postpone discussion of the *NucSkip* machine for now, and start with the *NucMon* machine. The *NucMon* machine models the behaviour just described at an abstract level. At initialisation time, the temperature θ is assumed to be θ_0 and a heating period is about to start (i.e. the rod is out). The *INITIALISATION* enables the *WaitRodOut* pliant event which lasts for the duration of the heating period. This specifies that the temperature is to increase in a monotonic fashion, using the MONINC modality, and furthermore states that the initial and final temperature values over the heating period differ by Δ.[3]

The significance of the latter is that, on the basis of monotonicity, we can deduce that the temperature throughout the heating period remains less than the final value attained during the period, $\theta(\mathbb{t}_{\mathrm{L}}) + \Delta$. This in turn is sufficient to guarantee that the invariant $\theta \leq \Theta_{\mathrm{TOL}}$ is satisfied throughout the heating period provided that $\theta_0 + \Delta \leq \Theta_{\mathrm{TOL}}$ holds, where Θ_{TOL} is the maximum tolerable reactor temperature that still assures safety. Since we have not included this last condition in the model, it would emerge as a missing hypothesis in any attempt to prove the pliant event invariant preservation PO (4) for *WaitRodOut*.

Similar arguments apply for the remainder of the behaviour of the *NucMon* machine. Thus, after 3 time units, mode event *RodIn* is enabled and preempts pliant event *WaitRodOut*. *RodIn* changes the *rod* variable from *out* to *in*, which enables *WaitRodIn*. In line with our simple modelling, the *WaitRodIn* pliant event exactly reverses the effect of *WaitRodOut*. The behaviour of *WaitRodIn* is also specified using a modality. This time it is the MONDEC modality, and it is again stated that the initial and final tempertaure values over the heating period differ by Δ. This time monotonicity guarantees

[3] In the *WaitRodOut* event, the final temperature value is referred to as $\theta(\mathbb{t}_{\mathrm{R}})$. This is slightly incorrect technically, since the actual value of θ at the time point referred to by $\mathbb{t}_{\mathrm{R}}$ falls outside the left closed right open interval $[\mathbb{t}_{\mathrm{L}} \ldots \mathbb{t}_{\mathrm{R}})$ which is the actual period during which any execution of *WaitRodOut* is active. A more technically correct reference to the final temperature reached would be $\mathsf{LLIM}(\theta(\mathbb{t}_{\mathrm{R}}))$, where LLIM is an additional modality that refers to the left limit of the expression enclosed. Instead of using such machinery, we have implicitly used a convention stipulating that any reference to a value at $\mathbb{t}_{\mathrm{R}}$ is in fact a reference to the relevant left limit value. This convention is to be understood as applying throughout Fig. 2. (Actually, since the temperature θ is always continuous in the behaviour specified in the *NucMon* machine, there is no difference between the actual $\mathbb{t}_{\mathrm{R}}$ value, and the corresponding left limit value at $\mathbb{t}_{\mathrm{R}}$, since no mode event of *NucMon* alters θ discontinuously.)

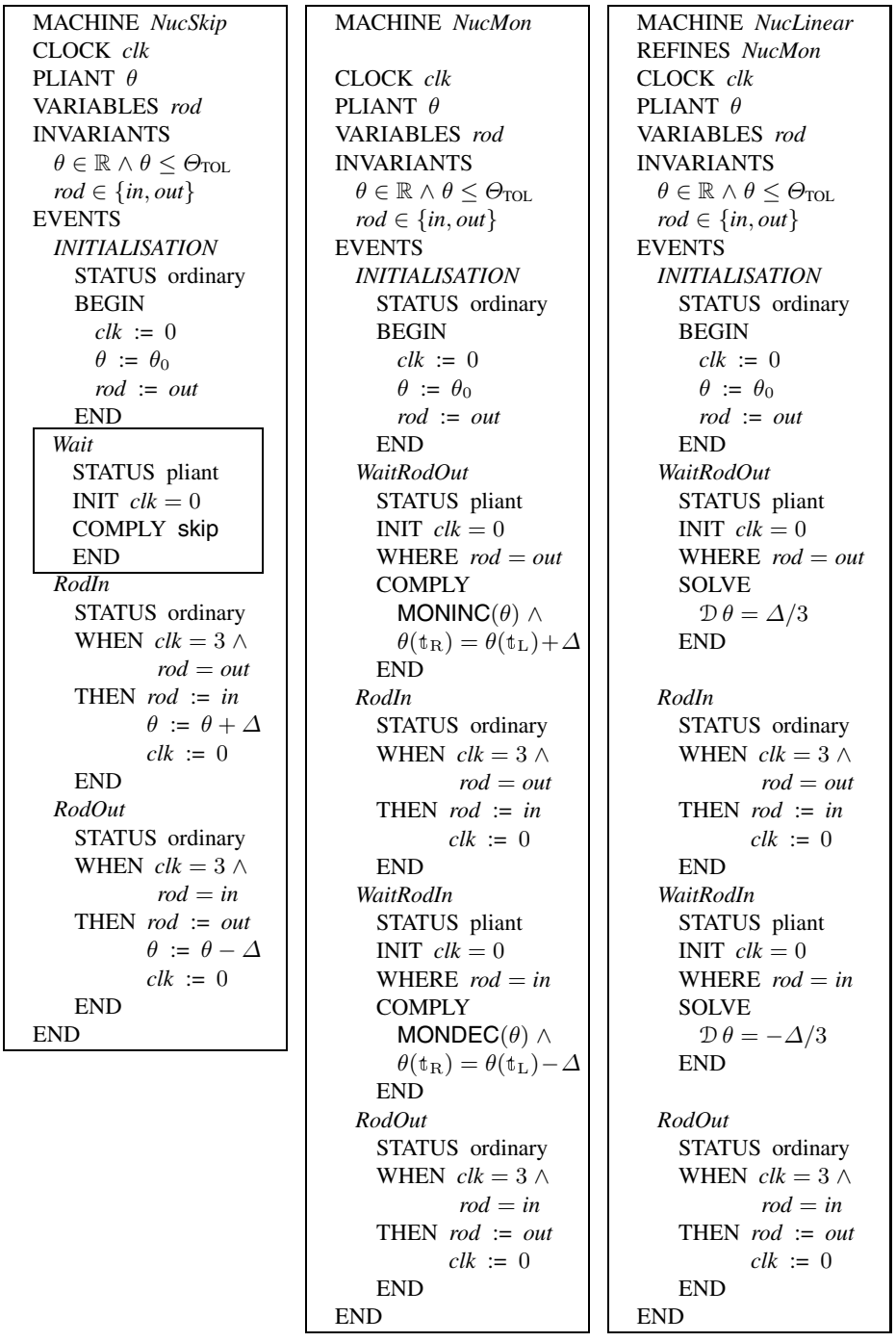


Fig. 2. Hybrid Event-B machines for the nuclear reactor case study

that the temperature throughout the cooling period remains less than the initial value at the beginning of the period, $\theta(\mathbb{t}_{\mathrm{L}})$.

After 3 time units, mode event *RodOut* is enabled and preempts *WaitRodOut*. Because of our particularly simple modelling, *RodOut* returns the machine to exactly the state at intialisation, and the whole cycle of behaviour repeats indefinitely.

We now turn to machine *NucLinear*. This machine refines *NucMon*. Mostly it is identical to *NucMon*. To save space, we have not included the REFINES clauses for the individual events — it is to be assumed that each event of *NucLinear* refines the similarly named event in *NucMon*. The only differences between the two machines are in the *WaitRodOut* and *WaitRodIn* pliant events. While these events are specified loosely in the *NucMon* machine via modalities that admit an uncountable infinity of realisations, in the *NucLinear* machine, their behaviours are defined deterministically, via the differential equations $\mathcal{D}\,\theta = \pm\Delta/3$. Since the solutions of these, $\theta(t - \mathbb{t}_{\mathrm{L}}) = \theta(\mathbb{t}_{\mathrm{L}}) \pm (\Delta/3)(t - \mathbb{t}_{\mathrm{L}})$ are obviously monotonic, and also satisfy the properties needed at $\mathbb{t}_{\mathrm{R}} = \mathbb{t}_{\mathrm{L}} + 3$, they satisfy the specifications of *WaitRodOut* and *WaitRodIn* in *NucMon*, and hence we will be able to discharge the correctness PO for pliant event refinement (5) for these events, which is the only non-identity part of the refinement.

Reexamining the preceding from a requirements perspective, it is reasonable to presume that the crucial elements of the two machines *NucMon* and *NucLinear* were designed at least partly in tandem. The abstract specifications of pliant events *WaitRodOut* and *WaitRodIn* in *NucMon* must have been designed with at least a good degree of confidence that the rate of increase/decrease of temperature that they demanded was feasible, i.e. that realisations via the detailed behaviours of *WaitRodOut* and *WaitRodIn* in the *NucLinear* machine were achievable using the physical apparatus available.

Now we turn to the *NucSkip* machine. Its aim is to capture as much as possible of the system behaviour within mode events. We see this in the fact that the counterparts of the two pliant events *WaitRodOut* and *WaitRodIn* in the other two machines are required merely to skip in this one (over an extended time period), something which warrants the two events being combined into a single *Wait* event (whose somewhat superfluous nature is highlighted by the box surrounding it). Along with *Wait* just skipping, the two mode events *RodIn* and *RodOut* take on the additional job of recording the movements in temperature via increments of $\pm\Delta$.

In fact, with a suitably designed (and nontrivial) retrieve relation, the *NucSkip* and *NucMon* machines are interrefinable. To see this, we would have to rename the variables in the two machines in order to properly define the retrieve relation. While we do not pursue this in full detail, we can point out the essentials as follows. The retrieve relation $R(\theta_{NucSkip}, \theta_{NucMon})$ (which is oriented so that *NucSkip* is the abstract model and *NucMon* is the concrete model) has to take θ_{NucMon} and relate it to either θ_0 or to $\theta_0 + \Delta$ according to the value of *rod*, thus:

$$\begin{aligned} K(\theta_{NucSkip}, \theta_{NucMon}) \;\equiv\; & \\ & (rod = out \wedge \theta_0 \le \theta_{NucMon} < \theta_0 + \Delta \wedge \theta_{NucSkip} = \theta_0) \vee \\ & (rod = in \wedge \theta_0 + \Delta \ge \theta_{NucMon} > \theta_0 \wedge \theta_{NucSkip} = \theta_0 + \Delta) \end{aligned} \tag{24}$$

The technically most interesting points regarding the refinement concern how the joint invariant $K(\theta_{NucSkip}, \theta_{NucMon})$ is preserved across the discontinuities at the mode events.

However, it is not hard to see that it all works out as required. For example, as an occurrence of *RodIn* approaches, θ_{NucMon} is slightly less than $\theta_0 + \Delta$, while $\theta_{NucSkip}$ is still θ_0, satisfying $K(\theta_{NucSkip}, \theta_{NucMon})$. Then, as soon as *RodIn* completes, θ_{NucMon} immediately starts to decrease from $\theta_0+\Delta$, while $\theta_{NucSkip}$ is now $\theta_0+\Delta$, again satisfying $K(\theta_{NucSkip}, \theta_{NucMon})$ because of the altered value of *rod*. Event *RodOut* is similar, and, filling in the remaining details, the claimed interrefinability of *NucSkip* and *NucMon* can be established.

The interrefinability allows us to regard *NucSkip* as either an abstraction or a refinement of *NucMon*. The mathematics of the refinement is in fact typical of digital implementation techniques, whereby continuous behaviours are implemented by discretized means, under benign enough conditions. However, in this case, the relatively long duration of the pliant events, makes viewing *NucSkip* as an *implementation* rather unconvincing.

The view that *NucSkip* is an abstraction is more productive. Besides allowing for the passage of time, the pliant events of *NucSkip* do nothing. Nevertheless, despite this relative triviality, whether or not the crucial invariant $\theta \leq \Theta_{\mathrm{TOL}}$ is preserved can still be determined from such a model. Note that the determination of this requires discrete computation exclusively, in contrast to making the same determination more directly from the *NucMon* or *NucLinear* machines, a potentially important simplification in the context of mechanical reasoning.

The *NucSkip* machine is a whisker away from a conventional Event-B machine. In fact, noting that the passage of time has no significance in this model, we could dispense with the pliant events completely, and be left with a genuine Event-B machine. In [17, 18, 4] the authors pursue a very similar approach in comparable examples. There, the abstract Event-B models just alluded to, are refined to more concrete Event-B models which handle the continuous behavours by packaging up pieces of continuous behavour in lambda abstractions, and assembling the overall behaviour as a succession of modifications that take place at the discrete transitions that correspond to our mode events. (This is patterned after the manner in which hybrid systems are modelled in the action systems framework [9, 7, 8].)

On the one hand, there is little difference between the approach we have developed here and the works of these other authors, if the aim is to explore the system's reachability relation and invariant preservation properties through the extreme values attained during the system's runs, these being computed using the discrete versions of the models. On the other hand, one consequence of using a (pure) discrete Event-B model for the management of these extreme values, is that the joint invariant defining its refinement to a more detailed model (taking real time and continuous behaviour into account) is typically restricted to holding only at the moments that the discrete events take place, i.e. pointwise at a few isolated moments of time, despite the fact that the continuous behaviour is active all the time. This observation severely weakens the connection between the models used, and the actual real world requirements that ought to be captured in the invariants.

To caricature the above in the context of our case study, it is no good (from the viewpoint of the real world requirements) if we are sure that the reactor temperature is safe at 3pm and at 4pm (because these are the times at which the discrete Event-B

model's events take place and so the invariant can be asserted) and yet the reactor core is able to suffer a meltdown at 3.30pm (because no event took place then, and so the invariant could not be asserted at this time).

In our approach, the modalities we have introduced earlier enable us to conveniently package up useful properties of the continuous behaviors of interest, in a way that allows us to reduce the maintenance of the invariants that we are concerned with to discrete computations very similar to those of the previous approaches, *while nevertheless retaining fully the ability to rigorously assert the invariants needed at all times relevant to the requirements, and not just at those times when discrete events take place*. In this sense we regard our approach as an improvement on the pure discrete Event-B approach, beyond simply offering a more direct treatment of hybrid behaviour.

The above discussion also highlights the fact that mere (inter)refinability alone can be a very weak relationship between system models, unless the refinement relationship is appropriately validated against the application requirements. This is particularly important when purely discrete models are being related to models which treat real time as an indispensable element. Therefore it is crucial that the content of any such refinement relationship is critically evaluated. Our case study illustrates this particularly well since the to-all-intents-and-purposes discrete model *NucSkip* bears almost no visible relationship to the more 'realistic' *NucLinear* model that purports to reflect the behaviour of the physical system in a recognisable way. After all, the physical temperature *does not* undergo discontinuous changes when the control rod is inserted or withdrawn; and the physical temperature *does not* remain invariant in the time intervals between insertions/withdrawals. So if we are to use such a discrete model, it is vital that we can relate its behaviour to more realistic models via strong and convincing invariants.

10 Conclusions

In the previous sections we briefly reviewed Hybrid Event-B, and then discussed the motivations for introducing pliant modalities into the user level language. Basically, these included readability for application designers, so that intuitively straightforward properties would not be masked by convoluted low level descriptions. Furthermore, the possibility of simplifying the challenge for mechanised reasoning systems when dealing with large applications, by raising the level of reasoning abstraction via these modalities, provided an additional motivating factor. The motivating discussion went on to consider the interesting issue of where and how in a refinement hierarchy, the requirements concerning continuously varying entities were to be addressed, given that these are invariably strongly linked to real world properties of physical apparatus.

We then introduced a selection of simple modalities for functions of time returning a single real value, illustrating one of them in a case study based on rising and falling temperature in a nuclear reactor. The intention was not to give an exhaustive list of all modalities that might ever possibly be useful, but to give a representative selection.

In general, we would expect that an automated environment for supporting Hybrid Event-B application development would have the capability of incorporating user defined modalities introduced to support specific application.

The connection with automated environments and mechanised reasoning systems deserves further consideration. One aspect that we have not described in detail in this

relatively brief paper, is that along with its definition, each modality needs a selection of properties that a verification environment for some application can readily make use of. In other words each modality needs to be supplied as a *theory* containing not only its definition, but also relevant properties.

Such properties need to be true of course. For very simple modalities such as those discussed in this paper, the properties of interest have been well known in the mathematics literature for around two centuries (see for example [16, 13, 12]). So full scale mechanical verification for these is perhaps superfluous. On the other hand, as the modalities introduced by users become more sophisticated and less conventional, the need for mechanical corroboration of their claimed properties becomes more pressing.

The mechanical corroboration issue raises an interesting challenge not present in the verification of purely discrete systems. This is that, whereas the *proof* of the properties of typical pliant modalities requires low level reasoning typical of arguments in mathematical analysis, usually needing a considerable number of interleavings of quantifications over higher order structures to be considered, the *use* of such properties in an application context is overwhelmingly symbolic and algebraic, needing perceptive strategies for equational substitution of equals for equals, and similar techniques. These are conflicting requirements for a verification system. The best proposal would therefore be to use different tools to address these different requirements: a system specifically geared to higher order reasoning and multiple nested quantifications at the low level, and a system more geared to decomposition and algebraic reasoning at the applications level. Such considerations pose an interesting challenge for any tool (eg. the Rodin Tool, [15]) that contemplates an extension to deal with the capabilities of Hybrid Event-B.

References

1. Report: Cyber-Physical Systems (2008), `http://iccps2012.cse.wustl.edu/_doc/CPS_Summit_Report.pdf`
2. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
3. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
4. Abrial, J.-R., Su, W., Zhu, H.: Formalizing Hybrid Systems with Event-B. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 178–193. Springer, Heidelberg (2012)
5. Back, R.J.R., Sere, K.: Stepwise Refinement of Action Systems. In: van de Snepscheut, J.L.A. (ed.) MPC 1989. LNCS, vol. 375, pp. 115–138. Springer, Heidelberg (1989)
6. Back, R.J.R., von Wright, J.: Trace Refinement of Action Systems. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 367–384. Springer, Heidelberg (1994)
7. Back, R.J., Cerschi Seceleanu, C.: Modeling and Verifying a Temperature Control System using Continuous Action Systems. In: Proc. FMICS 2000 (2000)
8. Back, R.J., Cerschi Seceleanu, C., Westerholm, J.: Symbolic Simulation of Hybrid Systems. In: Strooper, P., Muenchaisri, P. (eds.) Proc. APSEC 2002, pp. 147–155. IEEE Computer Society Press (2002)
9. Back, R.J., Petre, L., Porres, I.: Continuous Action Systems as a Model for Hybrid Systems. Nordic J. Comp. 8, 2–21 (2001)
10. Banach, R., Butler, M., Qin, S., Verma, N., Zhu, H.: Core Hybrid Event-B: Adding Continuous Behaviour to Event-B (2012) (submitted)

11. Barolli, L., Takizawa, M., Hussain, F.: Special Issue on Emerging Trends in Cyber-Physical Systems. J. Amb. Intel. Hum. Comp. 2, 249–250 (2011)
12. Haggarty, R.: Fundamentals of Mathematical Analysis. Prentice Hall (1993)
13. Lang, S.: Real and Functional Analysis. Springer (1993)
14. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer (2010)
15. Rodin: European Project Rodin (Rigorous Open Development for Complex Systems) IST-511599, http://rodin.cs.ncl.ac.uk/
16. Rudin, W.: Principles of Mathematical Analysis. McGraw-Hill (1976)
17. Su, W., Abrial, J.-R., Huang, R., Zhu, H.: From Requirements to Development: Methodology and Example. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 437–455. Springer, Heidelberg (2011)
18. Su, W., Abrial, J.-R., Zhu, H.: Complementary Methodologies for Developing Hybrid Systems with Event-B. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 230–248. Springer, Heidelberg (2012)
19. Sztipanovits, J.: Model integration and cyber physical systems: A semantics perspective. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, p. 1. Springer, Heidelberg (2011), http://sites.lero.ie/download.aspx?f=Sztipanovits-Keynote.pdf
20. Tabuada, P.: Verification and Control of Hybrid Systems: A Symbolic Approach. Springer (2009)
21. Wikipedia: Absolute continuity
22. Willems, J.: Open Dynamical Systems: Their Aims and their Origins. Ruberti Lecture, Rome (2007), http://homes.esat.kuleuven.be/~jwillems/Lectures/2007/Rubertilecture.pdf

A Relational Approach to an Algebraic Community: From Paul Erdős to He Jifeng

Jonathan P. Bowen

Department of Informatics, Faculty of Business
London South Bank University, London SE1 0AA, UK
jonathan.bowen@lsbu.ac.uk
www.jpbowen.com

Abstract. Scholarly advance depends on the interaction of researchers in a large number of overlapping communities in different disciplines (mathematics, computer science, etc.) and fields within these disciplines (e.g., algebra, formal methods, etc.). Now that academic publications are largely accessible on the Internet, these connections are directly available through a number of resources and visualization tools that are available online. Academic links are typically in the form of co-authors, citations, supervisor/student, etc., forming different types of relations between pairs of researchers. This paper explores these links with some specific examples, including visualization of these relationships and their formalization using the Z notation.

1 Introduction

The development and transmission of scientific knowledge has always relied on collaboration [18]. A scientific theory can be considered as a network (or graph) of questions (vertices) and answers (arcs) [15]. Logic underlies the correct reasoning needed for the valid development of scientific theories [6]. A Community of Practice (CoP) is a social science model that can be used as a framework to study the process of producing a Body of Knowledge (BoK) for a particular field, such as formal methods [3].

In recent years, the speed of transmission and the quantity of knowledge available has accelerated dramatically, especially with the advent of the Internet and specifically the World Wide Web. Whereas previously academic papers were published on paper in journals, conference proceedings, technical reports, books, etc., now all these means of communication can and often are done largely electronically online. The plethora of information is becoming indexed more and more effectively.

For example, Google, as well as indexing the Web in general, also has specific facilities for indexing academic publications through Google Scholar (`http://scholar.google.com`) and books in general though the more widely known Google Books facility (`http://books.google.com`). It has a very complete and up-to-date corpus of data available compared to other sources. Google also allows users to provide a personalized presentation of their publications, hand-corrected as needed. The automated trawling of publications and citations done by Google Scholar works well in general for publications with a reasonable number of citations, where multiple copies of citations allows automated refinement of the information. Typically, there is also a "long tail" of

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 54–66, 2013.

data on uncited or lesser cited publications, some of which can be spurious. For example, it often trawls programme committee data for conferences where all the programme committee members are considered to be authors.

2 Publication Metrics

With the increasing fashion of attempting to measure research output by governments, fuelled by limited resources for research funding, a number of metrics have been developed that aim to provide a measure of the importance of a researcher in their field. The simplest measure is a citation count, but this has a number of drawbacks. There is often a significant number of publications with small citation counts that have little influence in practice. Most researchers have a smaller number of key publications that have been more highly cited by their peers. Typically, it is these that are most significant. Some attempts to provide a better indication of an author's influence based on their cited publications are presented in this section.

Despite the drawbacks of Google Scholar, it provides one of the best online indicators of an author's "*h-index*" [12], which provides one of the most popular indications of an author's influence on other researchers. This measures the maximum number of publications by an author that have citations greater than or equal to that number. This can be modelled on any given set using the Z notation [16] as follows, using a bag (or multiset) to record the citation count for each item:

$$[X]$$

$$\text{h-index} : \text{bag}\, X \to \mathbb{N}$$

$$\text{h-index}\, [\![ ]\!] = 0$$

$$\forall\, b : \text{bag}\, X;\ x : X \mid x \in \text{dom}\, b \land b(x) = max(\text{ran}\, b) \bullet$$
$$\quad \text{h-index}\, b =$$
$$\quad\quad \textbf{if}\ b(x) < \#b$$
$$\quad\quad \textbf{then}$$
$$\quad\quad\quad \#b - 1$$
$$\quad\quad \textbf{else}$$
$$\quad\quad\quad \text{h-index}(\{x\} \lhd b) + 1$$

Note that in Z, bags are defined as $\text{bag}\, X == X \pfun \mathbb{N}_1$, a partial function from any set X to non-zero natural numbers. X can be used to represent cited publications, for example, mapped to the number of citations associated with each of these publications.

It should be noted that this measure needs to be treated with some caution since different academic fields have significant variation in their publication patterns. For example, computer scientists tend to have a much lower number of co-authors than some physicists, who may play a very small role in a highly cited paper.

The number of citations for each individual publication and the total for an author are also automatically calculated and displayed on an author's personalized Google Scholar page (if set up by the author). As well as the h-index, the simpler "*i10-index*" is provided, which is just the number of publications of an author that have at least 10 citations. This can be generalized to an arbitrary number of publications as follows:

$$[X]$$

$$\text{i-index} : \mathbb{N} \to (\text{bag}\ X \to \mathbb{N})$$

$$\forall\, n : \mathbb{N};\ b : \text{bag}\ X \bullet \text{i-index}\ n\ b = \#(b \rhd (1\,..\,n-1))$$

The $\rhd$ operator represents range anti-restriction of a relation above, resulting the the number of items in b with a count of n or more.

Microsoft Academic Search(`http://academic.research.microsoft.com`) provides a competing database of academic publications online, started at Microsoft's research laboratory in Beijing, China. While it is not as complete or up to date as Google Scholar, it does provide better visualization facilities and also allows any individual to submit corrections for the entire database. These are checked before that are incorporated, so there is a delay ranging from days to weeks in any submitted updates. Academic Search also calculates the "*g-index*" [5] for an individual, a refinement of the h-index that provides as arguably better indication of an author's influence. With this metric, very highly cited publications (perhaps a very significant book for example) are given more weight than is the case with the h-index. The most cited g papers must have at least g^2 citations in total:

$$[X]$$

$$\text{g-index} : \text{bag}\ X \to \mathbb{N}$$

$$\text{g-index}\ [\![\,]\!] = 0$$

$$\forall\, b : \text{bag}\ X;\ x : X \mid x \in \text{dom}\ b \land b(x) = max(\text{ran}\ b) \bullet \text{g-index}\ b = \textbf{if}\ \Sigma b < \#b * \#b\ \textbf{then}\ \#b - 1\ \textbf{else}\ \text{g-index}(\{x\} \lhd b) + 1$$

The Σ summation function can be defined as follows:

$$[X]$$

$$\Sigma : \text{bag}\ X \to \mathbb{Z}$$

$$\Sigma\, [\![\,]\!] = 0$$

$$\forall\, x : X;\ n : \mathbb{Z} \bullet \Sigma\{x \mapsto n\} = n$$

$$\forall\, b, c : \text{bag}\ X \mid \text{dom}\ b \cap \text{dom}\ c = \varnothing \bullet \Sigma(b \cup c) = \Sigma b + \Sigma c$$

3 An Example

In this paper, we take as an example, the leading Chinese computer scientist He Jifeng [7,14] (see Figure 1) of East China Normal University in Shanghai (see Figure 2),

illustrating some of the visualization facilities of Academic Search and formalizing some of the aspects and measures of the relationships of authors and papers through co-authorship and citations. These links form mathematical graphs [15] that can be modelled as mathematical relations. The Z notation [11,16] is convenient for presenting relations in a structured manner using "schemas".

Fig. 1. Publication and citation statistics for He Jifeng on Academic Search

Academic Search also lists the conferences and journals for each author in reverse order of publication count and the main keywords associated with the publications of an author (see Figure 3). For example, He Jifeng is particularly active in the Software Engineering Workshop (SEW), the oldest software engineering event in the world (see Figure 4).

4 Formal Model in Z

The data available in facilities link Google Scholar and Microsoft Academic Search can be formalized mathematically at an abstract level. Here, a formal model of some aspects is presented using the Z notation [16]. Z has a convenient schema notation that allows a model to be gradually built up based on previous mathematical descriptions and underlying given sets, here representing people and publications such as academic papers.

$$[PEOPLE, PAPERS]$$

Authors are people who write publications such as papers. These papers can cite a number of other papers as references, but not normally themselves. Papers that reference each other or even cycles of references are possible, although are not the normal case. Typically references are in the form of a list at the end of the document that can be extracted reasonably accurately and automatically by software. A given paper may have citations from other papers in their list of references.

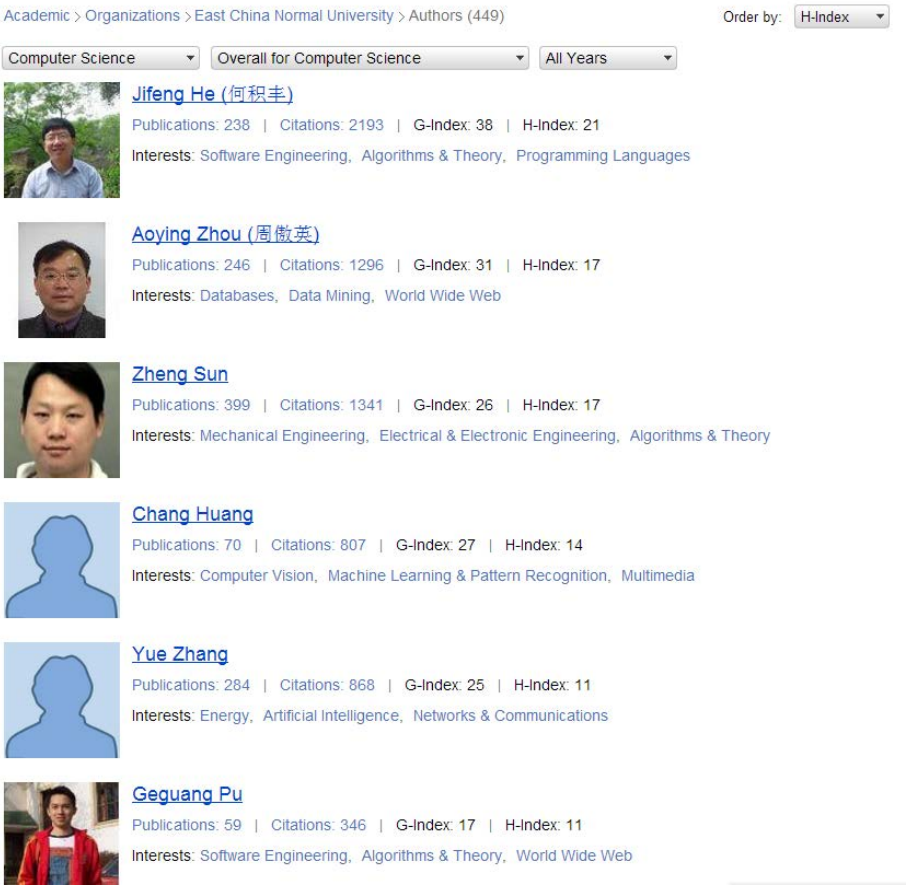


Fig. 2. Top computer science authors from East China Normal University on Academic Search

Relationships

$$authors : PEOPLE \leftrightarrow PAPERS$$
$$references, citations : PAPERS \leftrightarrow PAPERS$$

$$references \cap \mathrm{id}\, PAPERS = \varnothing$$
$$\mathrm{dom}\, references \subseteq \mathrm{ran}\, authors$$
$$citations = references^{\sim}$$

An author has a number of co-authors from all the publications that they have written and also a number of associated authors who have cited that author's publications in their own publications. An author cannot be their own co-author, but they can cite themselves.

Fig. 3. Top conferences, journals and keywords associated with He Jifeng's publications on Academic Search

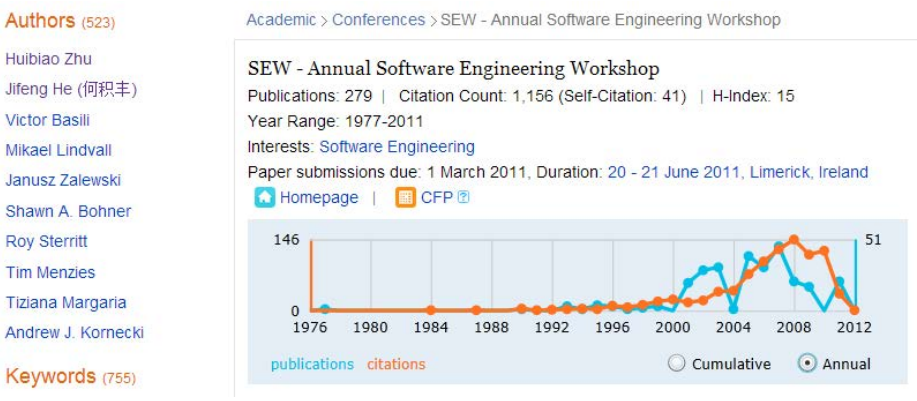


Fig. 4. Top authors and publication/citation statistics for the Software Engineering Workshop on Academic Search

Authors

$$
\begin{array}{l}
Relationships \\
coauthors, citingauthors : PEOPLE \leftrightarrow PEOPLE \\
\hline
coauthors = (authors \fatsemi authors^{\sim}) \setminus \mathrm{id}\, PEOPLE \\
citingauthors = authors \fatsemi citations \fatsemi authors^{\sim}
\end{array}
$$

Academic search allows the visualization of the main co-authors (see Figure 5) and the main citing authors (see Figure 6) for anyone in the database. Figure 5 represents a graphical view of a subset of the relation $\{author\} \lhd related \rhd coauthors(\!|\ \{author\}\ |\!)$ for a particular $author$ in the centre. Connections between co-authors are shown, as well as with the main author under consideration. Figure 6 shows a similar graphical view of a subset of the relation $\{author\} \lhd citingauthors$, again for a particular $author$ at the top left of the diagram.

If an author is a co-author of a particular author, then the reverse is also true.

$$\vdash coauthors = coauthors^{\sim}$$

Authors can be related to other authors transitively via co-authorship of a number of papers.

Related

$$
\begin{array}{l}
Authors \\
related : PEOPLE \leftrightarrow PEOPLE \\
\hline
related = coauthors^{+} \setminus \mathrm{id}\, PEOPLE
\end{array}
$$

It can be of interest to find the links transitively through co-authors for a pair of authors. The two author's collaborative distance can be measured as the minimum number of links needed to connect them through co-authorship. For direct co-authors, this is 1. This has led to the concept of a collaborative distance between two authors, via co-authorship of publications. For example, the 20th-century Hungarian mathematician Paul Erdős co-authored papers with over 500 people [4], probably making him the most prolific mathematical collaborator ever. The collaborative distance from Erdős is a measure of the level of involvement of a researcher in the field of mathematics. Since computer science is a related field, computer scientists, especially those working in more theoretical areas, are often quite closely related to Erdős through co-authorship too. The concept of the "Erdős number", the collaborative distance from Erdős, has emerged as a metric of involvement in mathematical research and also related disciplines like computer science [4]. Paul Erdős himself is deemed to have an Erdős number of 0. Other authors can be assigned a number that is the minimum length of the co-authorship path that links them with Erdős, assuming they are related by such a path of course.

$$erdos : PEOPLE$$

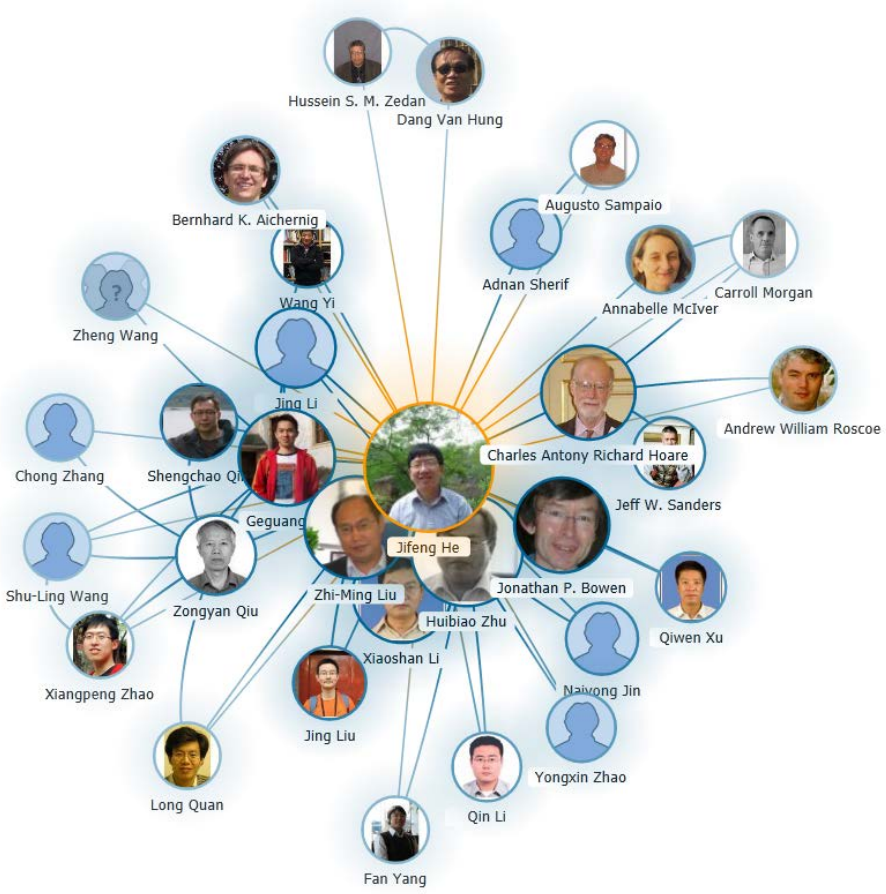


Fig. 5. Main co-authors of He Jifeng on Academic Search

$$
\begin{array}{|l}
\textit{ErdosNumber} \\
\textit{Related} \\
erdosnumber : PEOPLE \pfun \mathbb{N} \\
\hline
erdosnumber(erdos) = 0 \\
\forall\, author : PEOPLE \mid author \in related(\!|\, \{erdos\} \,|\!) \bullet \\
\quad erdosnumber(author) = \\
\qquad min(erdosnumber(\!|\, coauthors(\!|\, \{author\} \,|\!) \,|\!)) + 1
\end{array}
$$

All co-authors of Erdős have an Erdős number of 1.

$$\vdash erdosnumber(\!|\, coauthors(\!|\, \{erdos\} \,|\!) \,|\!) = \{1\}$$

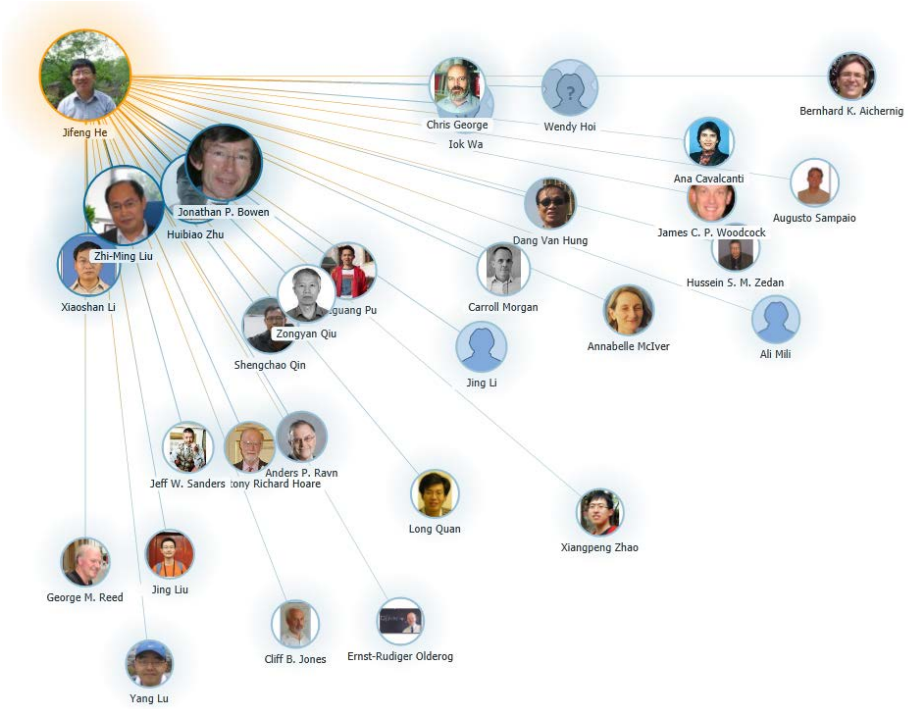


Fig. 6. Main citing authors for He Jifeng on Academic Search

Authors who have written publications with co-authors of Erdős have an Erdős number of 2. This can be repeated iteratively, following the path of minimum length when there is more that one path.

Academic Search provides a visualization of a selecton of the shortest paths between any two co-authors, with Paul Erdős as the default second author. See Figure 7 for an example. Here the minimum distance between the selected author and Erdős via author nodes in the graph is three, so the author has an Erdős number of 3.

Only some people are published authors. These people and the publications they have produced are of special interest academically:

$$
\begin{array}{|l}
\hline
PublishedPapers \\
\hline
ErdosNumber \\
published : \mathbb{F}\, PEOPLE \\
papers : \mathbb{F}\, PAPERS \\
\hline
published = \mathrm{dom}\ authors \\
papers = \mathrm{ran}\ authors \\
\hline
\end{array}
$$

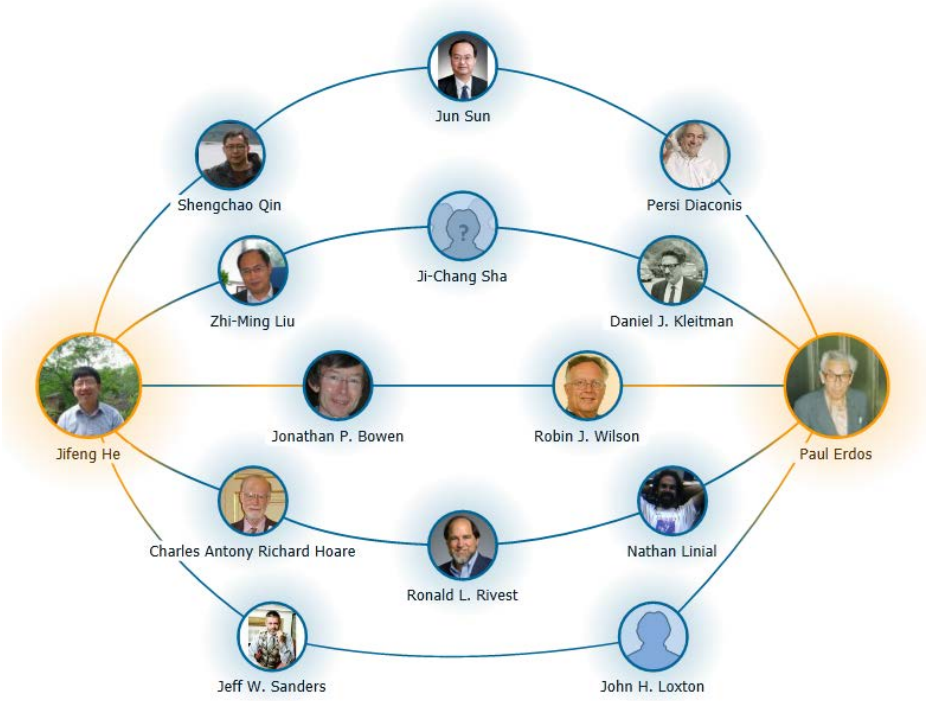


Fig. 7. Connections with Paul Erdős for He Jifeng on Academic Search

The publications of a specific author can be of particular interest too:

PapersBy

$$
\begin{array}{l}
PublishedPapers \\
papersby : PEOPLE \twoheadrightarrow \mathbb{F}\, PAPERS \\
\hline
\forall\, author : PEOPLE \mid author \in published \bullet \\
\quad papersby(author) = authors(\!|\, \{author\} \,|\!)
\end{array}
$$

The number of citations that a paper has received is also of interest as a rough metric for the importance of that paper.

NoOfCitations

$$
\begin{array}{l}
PapersBy \\
noofcitations : PAPERS \twoheadrightarrow \mathbb{N} \\
\hline
\forall\, p : PAPERS \mid p \in \mathrm{dom}\, citations \bullet \\
\quad noofcitations(p) = \#(citations(\!|\, \{p\} \,|\!))
\end{array}
$$

Papers with a non-zero citation count for a particular author are of special interest.

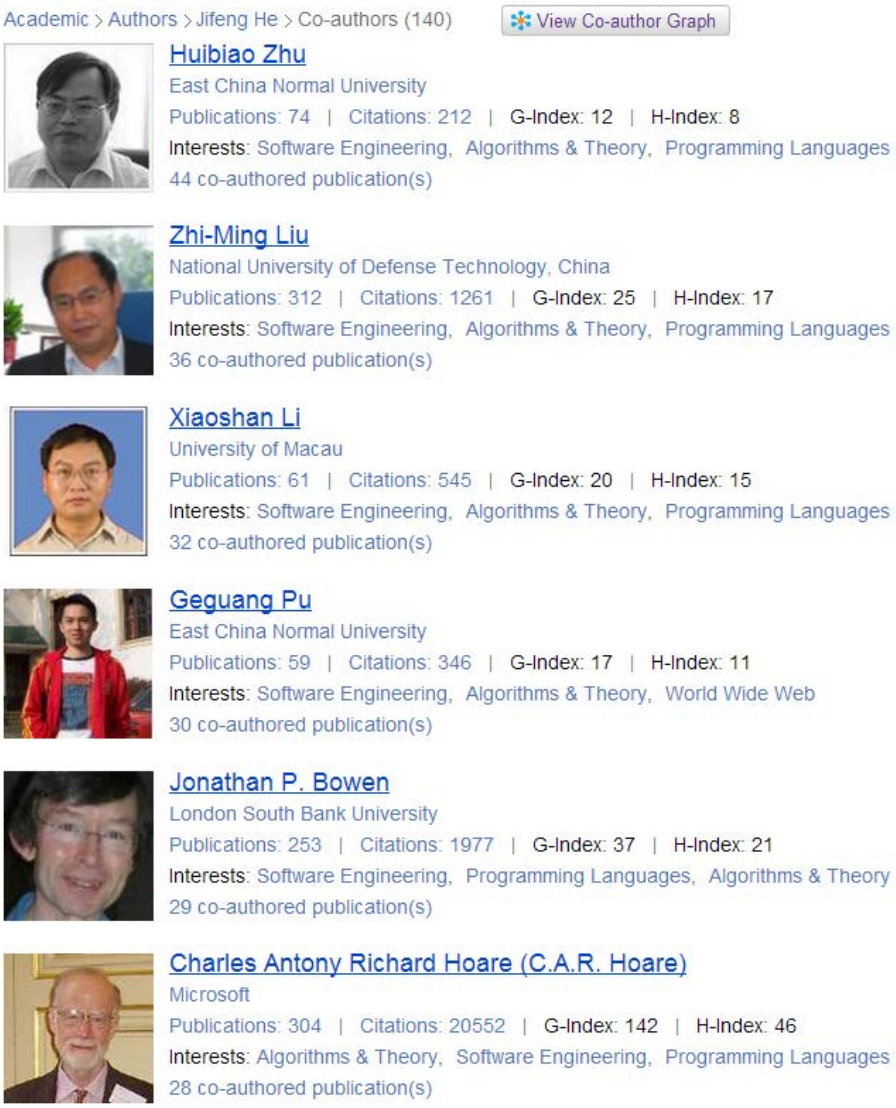


Fig. 8. Top co-authors with He Jifeng as listed on Academic Search

$$
\begin{array}{|l}
\hline
AuthorsCitedPapers \\
NoOfCitations \\
authorscitedpapers : PEOPLE \pfun \text{bag } PAPERS \\
\hline
\forall\, author : PEOPLE \mid author \in published \bullet \\
\quad authorscitedpapers(author) = papersby(author) \rhd noofcitations \rhd \{0\} \\
\hline
\end{array}
$$

The h-index, g-index, and i10-index are all metrics for an author's influence in their field, although interpretation of their significance depends on various factors, including the nature of publishing patterns in the author's discipline, the length of time the author has been active, etc. These can be derived for a given author from the previously presented generic definitions:

Indexes

$$AuthorsCitedPapers$$
$$Hindex, Gindex : PEOPLE \pfun \mathbb{N}$$
$$Iindex : \mathbb{N} \to (PEOPLE \pfun \mathbb{N})$$

$$Hindex = authorscitedpapers \fatsemi \text{h-index}$$
$$Gindex = authorscitedpapers \fatsemi \text{g-index}$$
$$\forall i : \mathbb{N} \bullet Iindex\ i = authorscitedpapers \fatsemi \text{i-index}\ i$$

5 Conclusions

This paper has presented visualization and formalization of academic collaboration on publications, using the online Academic Search facility and the Z notation respectively. A number of metrics have been covered including the Erdős number, h-index, g-index, and i10-index. Other metrics could easily be formalized if desired.

Z has proved to be an elegant formalism for capturing precise descriptions of various aspects of academic collaboration through co-authorship and citation of publications. This information is increasingly readily available online, allowing convenient exploration and visualization of these relationships and publication metrics. The formalization could easily be augmented if required to capture further details such as temporal information using the year of publication of papers, supervisor relationships for doctoral students, etc.

Acknowledgements. Jonathan Bowen is grateful for financial support from East China Normal University and Museophile Limited. Thank you to Microsoft for the Academic Search facility, which provided the screenshots for the figures in this paper. Many thanks especially to He Jifeng [1,7,10,13] and Zhu Huibiao [19,20,21,22,23] (see Figure 8) for many years of collaboration and inspiration, from the **ProCoS** project on Provably Correct Systems [2,9] onwards. The Z notation in this paper has been type-checked using the *f*UZZ type-checker [17]

References

1. Bowen, J.P., He, J., Pandya, P.: An approach to verifiable compiling specification and prototyping. In: Deransart, P., Małuszyński, J. (eds.) PLILP 1990. LNCS, vol. 456, pp. 45–59. Springer, Heidelberg (1990)
2. Bowen, J.P., Hoare, C.A.R., Langmaack, H., Olderog, E.-R., Ravn, A.P.: A ProCoS II Project final report: ESPRIT Basic Research project 7071. Bulletin of the European Association for Theoretical Computer Science (EATCS) 59, 76–99 (1996)
3. Bowen, J.P., Reeves, S.: From a Community of Practice to a Body of Knowledge: A case study of the formal methods community. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 308–322. Springer, Heidelberg (2011)

4. Bowen, J.P., Wilson, R.J.: Visualising virtual communities: From Erdős to the arts. In: Dunn, S., Bowen, J.P., Ng, K. (eds.) EVA London 2012 Conference Proceedings, Electronic Workshops in Computing (eWiC), pp. 238–244. British Computer Society (2012), arXiv:1207.3420v1
5. Egghe, L.: Theory and practise of the g-index. Scientometrics 69(1), 131–152 (2006), doi:10.1007/s11192-006-0144-7
6. Harré, R.: The Philosophies of Science: An Introductory Survey. Oxford University Press (1972)
7. He, J.: Provably Correct Systems: Modelling of Communication Languages and Design of Optimized Compilers. McGraw-Hill (1994)
8. He, J., Bowen, J.P.: Specification, verification and prototyping of an optimized compiler. Formal Aspects of Computing 6(6), 643–658 (1994)
9. He, J., et al.: Provably correct systems. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994. LNCS, vol. 863, pp. 288–335. Springer, Heidelberg (1994)
10. He, J., Page, I., Bowen, J.P.: Towards a provably correct hardware implementation of Occam. In: Milne, G.J., Pierre, L. (eds.) CHARME 1993. LNCS, vol. 683, pp. 214–225. Springer, Heidelberg (1993)
11. Henson, M.C., Reeves, S., Bowen, J.P.: Z logic and its consequences. CAI: Computing and Informatics 22(4), 381–415 (2003)
12. Hirsch, J.E.: An index to quantify an individual's scientific research output. Proceedings of the National Academy of Sciences 102(46), 16569–16572 (2005), arXiv:physics/0508025, doi:10.1073/pnas.0507655102
13. Hoare, C.A.R., He, J., Bowen, J.P., Pandya, P.: An algebraic approach to verifiable compiling specification and prototyping of the ProCoS Level 0 programming language. In: Directorate-General XIII of the Commission of the European Communities (eds.) ESPRIT 1990 Conference Proceedings, Brussels, Belgium, November 12-15, pp. 804–818. Kluwer Academic Publishers (1990)
14. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall Series in Computer Science (1998)
15. Sanitt, N.: Science as a Questioning Process. Institute of Physics Publishing, Bristol and Philadelphia (1996)
16. Spivey, J.M.: The Z Notation: A reference manual (Originally published by Prentice Hall, 2nd edn. 1992) (2001), `http://spivey.oriel.ox.ac.uk/mike/zrm/`
17. Spivey, J.M.: The *f*UZZ type-checker for Z (2008), `http://spivey.oriel.ox.ac.uk/mike/fuzz/`
18. Van Doren, C.: A History of Knowledge: Past, present, and future. Ballantine Books, New York (1991)
19. Zhu, H., Bowen, J.P., He, J.: From operational semantics to denotational semantics for verilog. In: Margaria, T., Melham, T. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 449–464. Springer, Heidelberg (2001)
20. Zhu, H., Bowen, J.P., He, J.: Soundness, completeness and non-redundancy of operational semantics for Verilog based on denotational semantics. In: George, C.W., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 600–612. Springer, Heidelberg (2002)
21. Zhu, H., He, J., Bowen, J.P.: From algebraic semantics to denotational semantics for Verilog. Innovations in Systems and Software Engineering: A NASA Journal 4(4), 341–360 (2008)
22. Zhu, H., He, J., Li, J., Bowen, J.P.: Algebraic approach to linking the semantics of web services. Innovations in Systems and Software Engineering: A NASA Journal 7(3), 209–224 (2011)
23. Zhu, H., Yang, F., He, J., Bowen, J.P., Sanders, J.W., Qin, S.: Linking operational semantics and algebraic semantics for a probabilistic timed shared-variable language. The Journal of Logic and Algebraic Programming 81(1), 2–25 (2012)

Practical Theory Extension in Event-B

Michael Butler[1] and Issam Maamria[2]

[1] Electronics and Computer Science, University of Southampton, UK
[2] UBS, UK

Abstract. The Rodin tool for Event-B supports formal modelling and proof using a mathematical language that is based on predicate logic and set theory. Although Rodin has in-built support for a rich set of operators and proof rules, for some application areas there may be a need to extend the set of operators and proof rules supported by the tool. This paper outlines a new feature of the Rodin tool, the theory component, that allows users to extend the mathematical language supported by the tool. Using theories, Rodin users may define new data types and polymorphic operators in a systematic and practical way. Theories also allow users to extend the proof capabilities of Rodin by defining new proof rules that get incorporated into the proof mechanisms. Soundness of new definitions and rules is provided through validity proof obligations.

1 Introduction

Abrial's Event-B is a formalism for refinement-based development of discrete event systems [1]. Its deployment is supported by the Rodin toolset which includes proof obligation generation and verification through a collection of mechanical provers [2]. An Event-B machine consists of a collection of variables, invariants on those variables and a collection of guarded events that may update the machine variables. An Event-B development consists of a collection of machines linked by refinement and refinement is verified through proof obligations for preservation of gluing invariants between abstract and concrete variables. Abrial's book [1] contains a range of refinement case studies in Event-B and many other Event-B case studies have been undertaken by academic researchers (e.g., `wiki.event-b.org/index.php/Event-B_Examples`) and by industry (e.g.[17]).

In Event-B, types, axioms, invariants, guards and actions may be defined using a set-theoretic mathematical language. While the mathematical language supported by the Rodin tool is rich (including operators on integers, sets, relations and functions), there is always a need to extend the mathematical language to broaden further the expressivity of the modelling notation. Because proof plays such a central role in the Event-B approach, hand-in-hand with any new mathematical operator definitions, there is a need to support proofs involving those operators. The need for an extensible mathematical language and theories in Event-B was envisaged by Abrial [3] where the need for a generic extension mechanisms, as found in languages such as PVS [15] and Isabelle [14], was identified. We refer to the process of defining new mathematical types and operators, together with associated proof rules, as *theory extension*.

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 67–81, 2013.

As well as supporting a rich mathematical language, the Rodin tool provides a range of automatic and interactive mechanical provers for proving obligations expressed in that language. In the earlier releases of Rodin, there were no mechanisms available for users to define new operators nor to extend the mechanical provers with new proof rules. Such extensions could only be undertaken by modifying the tool itself. This paper outlines recent work that overcomes this limitation, making theory extension part of the modelling process that can be undertaken in a systematic way without having to modify the Rodin tool. This has been achieved by adding a major new construct, the *Theory component*, to Event-B. This feature dis available in Rodin as a plug-in[1].

This paper outlines the theory component, how it enables the process of theory extension in the Event-B language and how it is supported in the Rodin tool. By allowing commonly-occurring structures to be captured as generic types, operators and proof rules, we allow these structures to be reused, thus reducing modelling and proof effort in the longer term. Genericity is achieved by supporting the definition of polymorphic operators and proof rules. These polymorphic operators and rules are instantiated with more specific types in modelling and proof, e.g., an operator with an argument of type $\mathbb{P}(\alpha)$ is instantiated with an argument of type $\mathbb{P}(\mathbb{Z})$.

In the earlier releases of Rodin, types were defined using set theory (power set and cartesian product) and there was no support for inductive data types (such as lists or trees). The new theory component supports the definition of inductive data types, along with recursive operator definitions and proof by induction.

It is important that any theory extensions are sound. Verifying soundness of theories is achieved through the definition of soundness proof obligations. When a modeller defines a new theory, soundness proof obligations are generated and then proved with the existing Rodin framework. This follows the standard Event-B approach where consistency of models and correctness of refinement between models is verified by discharging standard proof obligations.

This paper is structured around the main elements that may be contained in a theory, namely, operator definitions, datatype definitions, rewrite rules and inference rules. After presenting the theory component, we address important related work on proof in Event-B and mechanised theorem proving in general (Section 10). We start with a brief overview of the core Event-B mathematical language.

2 Event-B Mathematical Language

In the Event-B mathematical language, *expressions* and *predicates* are separate syntactic categories. Expressions are defined using literals (e.g., 1), constants, variables and polymorphic operators (e.g., set union). Expression operators can have expressions as arguments – such an operator *op* with arguments x_1 to x_n is written in the form $op(x_1, \ldots, x_n)$. Operators can also have predicates as

[1] `wiki.event-b.org/index.php/Theory_Plug-in`

arguments, e.g., $(\lambda x \cdot P(x) \mid E(x))$ where $P(x)$ is a predicate and $E(x)$ is a expression.

Predicates, on the other hand, are built from predicate operators (e.g., $\in$, $\subseteq$), logical connectives and quantifiers. Predicate operators take expressions as arguments e.g., $x \in S$ has x and S as arguments.

Expressions have a *type* and we use α to denote types. Types are constructed as follows:

1. a basic set such as $\mathbb{Z}$ or a carrier set defined in an Event-B context,
2. a power set of a type, written $\mathbb{P}(\alpha)$,
3. a cartesian product of two types, written $\alpha_1 \times \alpha_2$.

The type of a expression operator *op* with arguments $x_1 \ldots x_n$ is defined using typing rules of the form:

$$\frac{\mathbf{type}(x_1) = \alpha_1 \ ... \ \mathbf{type}(x_n) = \alpha_n}{\mathbf{type}(op(x_1, ..., x_n)) = \alpha}.$$

Arguments of a basic predicate must satisfy typing rules, e.g., the typing rule for the basic predicate $finite(R)$ is:

$$\mathbf{type}(R) = \mathbb{P}(\alpha).$$

Note that types can be used as set expressions within the Event-B mathematical language, e.g., $\mathbb{Z}$ is both a type and a set expression. Furthermore, the type operators ($\mathbb{P}$ and $\times$) have a second role as operators on sets. For example, suppose T is a basic type and S is a subset of T, then the expression $\mathbb{P}(S)$ is a valid expression. For set expressions S and R, we have

$$\begin{aligned} R \in \mathbb{P}(S) \quad &\Leftrightarrow \quad R \subseteq S \\ o_1 \mapsto o_2 \in S \times R \quad &\Leftrightarrow \quad o_1 \in S \ \wedge \ o_2 \in R \end{aligned}$$

Alongside typing rules, expression operators have *well-definedness* conditions. $\mathbf{WD}(E)$ is used to denote the well-definedness predicate of expression E. Proof obligations are generated (if necessary) to establish the well-definedness of expressions appearing in models. To illustrate, we consider the expression $card(E)$ for which we have:

$$\mathbf{WD}(card(E)) \quad \Leftrightarrow \quad \mathbf{WD}(E) \ \wedge \ finite(E).$$

Functions versus operators. It is instructive to consider the relationship between operators and function application in Event-B. An Event-B function $f \in A \nrightarrow B$ is a special case of a set of pairs so the type of f is $\mathbb{P}(\mathbf{type}(A) \times \mathbf{type}(B))$. The functionality of f is an additional property defined by a predicate specifying a uniqueness condition:

$$\forall x, y, y' \cdot x \mapsto y \in f \ \wedge \ x \mapsto y' \in f \ \Rightarrow \ y = y'$$

The domain of f, written $dom(f)$, is the set $\{ x \mid \exists y \cdot x \mapsto y \in f \}$. Application of f to x is written $f(x)$ which is well-defined provided $x \in dom(f)$.

It is important to note that f is not itself an operator, it is simply a set expression. The operator involved here is implicit – it is the *function application* operator that takes two arguments, f and x. To make the operator explicit, function application could have been written as $apply(f, x)$, where *apply* is the operator and f and x are the arguments. However, in the Rodin tool, the shorthand $f(x)$ must be used.

Variables in the mathematical language are typed by set expressions. This means, for example, that a variable may represent a function since a function is a special case of a set (of pairs). Variables may not represent expression operators or predicates in the mathematical language. This means that, while we can quantify over sets (including functions), we cannot quantify over operators or predicates in Event-B.

3 Theory Component

Models in Event-B are specified by means of *contexts* (static properties of a model) and *machines* (dynamic properties of a model). A theory is a new kind of Event-B component for defining theories that may be independent of any particular models. A theory is the means by which the mathematical language and mechanical provers may be extended.

We describe the overall structure of Event-B theories. A theory component has a name, a list of global type parameters (global to the theory), and an arbitrary number of definitions and rules:

theory *name*
type parameters $T_1, \ldots, T_n$

$\{$ $\langle$ *Predicate Operator Definition* $\rangle$
| $\langle$ *Expression Operator Definition* $\rangle$
| $\langle$ *Data Type Definition* $\rangle$
| $\langle$ *Rewrite Rule* $\rangle$
| $\langle$ *Inference Rule* $\rangle$ $\}$

An Event-B theory has a name which identifies it. A theory can have an arbitrary number of type parameters which are pair-wise distinct, in which case the theory is said to be polymorphic on its type parameters. In the following it is important to recall that the mathematical language has two syntactic categories, *predicates* and *expressions*. We look at each form of definition and rule in turn in the following sections.

4 Defining New Predicate Operators

A new Event-B polymorphic operator can be defined by providing the following information:

1. *Parser Information*: this includes the syntax, the notation (infix or prefix), and the syntactic class (expression or predicate).
2. *Type Checker Information*: this includes the types of the child arguments, and the resultant type if the operator is a expression operator.
3. *Prover Information*: this includes the well-definedness of the operator as well as its definition which may be used to reason about it.

A predicate operator defines a property on one or more expressions. For example, the predicate x *divides* y holds when x is an integer divisor of y. This predicate is defined in the following way:

predicate *divides*
infix
args $x : \mathbb{Z},\ y : \mathbb{Z}$
condition $x \in \mathbb{N} \ \wedge \ y \in \mathbb{N}$
definition $\exists a \cdot y = a \times x$

This declares a new operator *divides*. It is declared as infix with two arguments x and y both of type $\mathbb{Z}$. This declaration makes the predicate E *divides* F syntactically valid for integer expressions E and F. The condition specifies a well-definedness condition – in this case that x and y must be naturals ($\mathbb{N} \subseteq \mathbb{Z}$). The final clause provides the definition of x *divides* y. That is, we have

$$x \textit{ divides } y \quad \Leftrightarrow \quad \exists a \cdot y = a \times x$$

A new predicate operator may be infix or prefix. For example, if *divides* had been declared as prefix, then $\mathit{divides}(E, F)$ would become syntactically valid. An infix predicate must have exactly two arguments.

Though in the above case the arguments are typed with the predefined type $\mathbb{Z}$, in general arguments may be typed using some of the type parameters defined for the theory which makes the predicate polymorphic on those type parameters.

The general structure of a basic predicate definition is as follows:

predicate *Identifier*
(**prefix** | **infix**)
args $x_1 : \alpha_1,\ \ldots\ , x_n : \alpha_n$
condition $P(x_1, \ldots, x_n)$
definition $Q(x_1, \ldots, x_n)$

5 Defining New Expression Operators

While a predicate operator forms a predicate from a number of expressions, an operator forms an expression from a number of expressions. We consider an example involving the representation of sequences as functions whose domains are contiguous ranges of naturals starting at 1, i.e., functions from $(1..n) \rightarrow T$. The *seq* operator takes a set s and yields all sequences whose members are in s:

operator *seq*
prefix
args $s : \mathbb{P}(T)$
definition $\{ f, n \cdot f \in (1..n) \rightarrow s \mid f \}$

Here *seq* is declared to be a prefix operator with a single argument represented by s of type $\mathbb{P}(T)$. Since T is a type parameter, this means that *seq* is polymorphic on type T. The final clause defines the expression $seq(T)$ in terms of the existing expression language. The definition means we have that:

$$seq(s) = \{ f, n \cdot f \in (1..n) \rightarrow s \mid f \}$$

Note that the result type of an operator is inferred from the definition. In this case, the type of $seq(s)$ is $\mathbb{P}(\mathbb{Z} \leftrightarrow T)$, that is, a set of relations[2] between integers and the polymorphic type T. The following is an example of another prefix operator *size* that yields the size of a sequence:

operator *size*
prefix
args $m : \mathbb{Z} \leftrightarrow T$
condition $m \in seq(T)$
definition $card(m)$

Here, the well-definedness condition is stronger than the type declaration on m, requiring that m is an element of $seq(T)$.

Proof obligations are generated to verify the well-definedness of definitions. The validity proof obligation for operator definitions ensures that expressions involving that operator are well-defined. The proof obligation specifies that, assuming the arguments are well-defined and the explicit well-definedness condition for the operator holds, then the definition itself is well-defined. An important aspect of defining an operator is the well-definedness condition to be used. A simple strategy may use the well-definedness of the operator's direct definition. An advantage of a user-supplied condition is the possibility of strengthening well-definedness conditions to simplify proofs. In order to ensure that a supplied condition is in fact stronger than the default (i.e., the one inferred from the direct definition), proof obligations are generated.

For example, the definition of *size* leads to a proof obligation requiring that $card(m)$ is well-defined whenever $m \in seq(T)$. This is provable from the condition that m is a sequence since any element of $seq(T)$ has a finite domain and $card(m)$ is well-defined when m is finite (in Section 8 this is expressed as an inference rule).

Operators may be infix in which case they may be declared to be associative and commutative. For example, the concatenation operator on sequences, declared as follows, is associative:

[2] $\alpha_1 \leftrightarrow \alpha_2$ is shorthand for $\mathbb{P}(\alpha_1 \times \alpha_2)$

operator $\frown$
infix assoc
args $m : \mathbb{Z} \leftrightarrow T,\ n : \mathbb{Z} \leftrightarrow T$
condition $m \in seq(T) \ \wedge \ n \in seq(T)$
definition $m \ \cup \ \{ \ i, x \ \cdot \ i \mapsto x \in n \mid size(m) + i \mapsto x \ \}$

The general form of an operator definition is as follows:

operator *Identifier*
(**prefix** | **infix**) [**assoc**] [**commut**]
args $x_1 : \alpha_1, \ \ldots, \ x_n : \alpha_n$
condition $P(x_1, \ldots, x_n)$
definition $E(x_1, \ldots, x_n)$

A conditional expression of the form, $COND(p, e1, e2)$, may be used to define operators (p is a predicate while $e1$ and $e2$ are expressions). For example, the *max* operator, that yields the maximum of two integers, is defined using a conditional expression as follows:

operator *max*
infix assoc commut *// declare max to be associative and commutative*
type parameters T
args $x : \mathbb{Z}, \ y : \mathbb{Z}$
definition $COND(\ x \geq y \ , \ x \ , \ y \ )$

Declaring an operator to be associative and commutative gives rise to proof obligations to verify these properties. Since the Rodin provers automatically make use of commutativity and associativity properties of operators, to avoid circular proofs, the proof obligations must be specified in terms of the operator definition rather than the operator itself, e.g., the above declaration of *max* will give rise to the following commutativity proof obligation:

$$COND(\ x \geq y \ , \ x \ , \ y \ ) \quad = \quad COND(\ y \geq x \ , \ y \ , \ x \ )$$

6 Defining New Datatypes

A new datatype declaration defines a new type constructor together with constructor and destructor functions for elements of the new type. For example the usual inductive list type constructor is defined as follows:

datatype *List*
type args T
constructors
nil
$cons(\ head : T, \ tail : List(T)\ )$

This defines

- A new type constructor *List*. $List(T)$ becomes a type for any type T.
- A set operator *List*. $List(s)$ is a set expression – the set of lists whose members are in set s
- Two constructors *nil* and *cons*
- Two destructors *head* and *tail*
- An induction principle on *List*

The general form of an inductive data definition is as follows:

datatype *Ident*
  type args $T_1 \ldots T_n$
  constructors
    $c_1(\ d_1^1 : \alpha_1^1,\ \cdots,\ d_1^j : \alpha_1^j\ )$
    $\vdots$
    $c_m(\ d_m^1 : \alpha_m^1,\ \cdots,\ d_m^k : \alpha_m^k\ )$

Constructor and destructor names must be distinct. Types in Event-B are assumed to be non-empty, and this must hold for datatypes. As such, each newly defined datatype must have a base constructor, i.e., a constructor that does not refer to the datatype being defined. Here each α_j^i is a type that may include occurrences of the type being defined $Ident(T_1 \ldots T_n)$. If α_j^i does include occurrences of $Ident(T_1 \ldots T_n)$, then α_j^i must be *admissible*, i.e., α_j^i is $Ident(T_1 \ldots T_n)$ or is formed from a cartesian product or an existing inductive data type. Without the admissibility check, the datatype cannot be constructed. In the context of Event-B, the admissibility check rules out the following datatype definition

$$t(\alpha) \quad ::= \quad C_1 \ \mid\ C_2(\mathbb{P}(t))$$

since there is no injective function of type $\mathbb{P}(t) \to t$ by Cantor's theorem.

Proof by induction is supported in Rodin though a special reasoner that generates an induction scheme for any particular hypothesis or goal of a proof.

6.1 Pattern Matching with Datatypes

When defining basic predicates and operators on inductive types, the usual pattern matching may be used. For example the *size* function on inductive lists is defined as follows:

operator *size*
  prefix
  args $a : List(T)$
  definition

match a	
nil	0
$cons(x, b)$	$1 + size(b)$

Since a is of type $List(T)$ the argument a may be matched against each of the constructors for $List$.

predicate *member*
prefix
args $x : T,\ a : List(T)$
definition

match a	
nil	$false$
$cons(y, b)$	$x \neq y \Rightarrow member(x, b)$

Pattern matching and conditional expressions can be used together. Here is an example of an operator definition that removes duplicates in a list and uses a conditional expression:

operator *remdup*
prefix
args a
condition $a \in List(T)$
definition

match a	
nil	nil
$cons(x, b)$	$COND(\ member(x, b)\ ,\ remdup(b)\ ,\ cons(x, remdup(b))\ )$

Type constructors as set operators. In Section 2 we stated that type constructors ($\mathbb{P}$ and $\times$) also serve as set expression operators. Data type constructors can also be used as set operators. For example, suppose S is a set expression of type $\mathbb{P}(T)$, then $List(S)$ is a set expression specifying the set of inductive lists whose elements all come from S. $List(S)$ satisfies the following properties:

$$nil \in List(S)$$
$$cons(x, t) \in List(S) \quad \Leftrightarrow \quad x \in S \ \wedge \ t \in List(S)$$

7 Rewrite Rules

A rewrite rule is used in automatic or interactive proof to rewrite an expression or predicate in order to faciliate proof. A rewrite involves a left hand pattern and one or more right hands. Each right hand may be guarded by some condition. For example, the following rewrite rule defines two ways of rewriting the expression $card(i..j)$ depending on a condition on i and j ($i..j$ is the set of integers between i and j):

rewrite *CardIntegerRange*
auto manual complete
vars $i : \mathbb{Z},\quad j : \mathbb{Z}$
lhs $card(i..j)$

rhs

$i \leq j$	$j - i + 1$
$i > j$	0

This rule states that $card(i..j)$ may be rewritten to $j-i+1$ if $i \leq j$ and rewritten to 0 if $i > j$. The above declaration means that the rewrite rule can be used in automatic and interactive proof modes. The 'complete' declaration means that the disjunction of the guards must be true. The variables of the rule (i and j) serve as meta variables that can be matched with any expression of the appropriate type.

The general form of a rewrite rule for expressions is as follows (where the lhs and rhs are expressions):

rewrite *Name*
[**auto**] [**manual**] [**complete**]
vars $x_1 : \alpha_1, \ \ldots, \ x_n : \alpha_n$
condition $P(x_1, \ldots, x_n)$
lhs $E(x_1, \ldots, x_n)$
rhs

$Q_1(x_1, \ldots, x_n)$	$E_1(x_1, \ldots, x_n)$
$\vdots$	$\vdots$
$Q_m(x_1, \ldots, x_n)$	$E_m(x_1, \ldots, x_n)$

A number of validity obligations are required to ensure the soundness of a rewrite rule:

- The conditions must be well-defined: $P \wedge WD(E) \ \Rightarrow WD(Q_i)$
- Each rhs must be well-defined: $P \wedge WD(E) \wedge Q_i \ \Rightarrow WD(E_i)$
- Each rhs must equal the lhs: $P \wedge Q_i \ \Rightarrow E = E_i$

In addition, if the rule is declared to be case complete, then a completeness condition is required ($P \ \Rightarrow \ Q_1 \vee \cdots \vee Q_m$).

The general form of a rewite for predicates is similar (with the lhs and rhs being predicates). The validity obligations are similar to those for expression rewrites.

8 Inference Rules

An inference rule has a list of hypothesis and a consequent. It is parameterised by one or more variables. For example, the following inference rule has two hypotheses and a consequent that may be inferred from the hypotheses:

rule *FiniteSeq*
vars $s : \mathbb{P}(T), \ m : \mathbb{Z} \leftrightarrow T$
given
$m \in seq(s)$
infer
$finite(m)$

Here is another inference rule showing that sequence concatenation is closed for elements of $seq(s)$:

rule *Concat1*
vars s, m, n
given $s \subseteq T$
$m \in seq(s)$
$n \in seq(s)$
infer $m \frown n \in seq(s)$

The general form of an inference rule is as follows:

rule *Name*
vars $x_1, \ldots, x_n$
given
$P_1(x_1, \ldots, x_n), \ \ldots, \ P_m(x_1, \ldots, x_n)$
infer
$Q(x_1, \ldots, x_n)$

A number of validity obligations are required to ensure the soundness of an inference rule:

- The rule must be well-defined: $WD(P_1 \land \cdots \land P_m) \ \Rightarrow WD(Q)$
- The rule must be provable: $P_1 \land \cdots \land P_m \ \Rightarrow Q$

Using Inference Rules. Inference rules can be used in a backward style as well forward style. If used in backward style, the prover discharges or splits the goal. If applied in a forward style, more hypotheses get generated.

9 Axiomatic Definitions

The constructs we have outlined so far in this paper allow for direct definitions of new operators, inductive data types and recursive definitions of new operators over inductive types. For some types and operators this is not enough. For example, theories of integers and reals are typically defined axiomatically. That is, the types are not inductive data types, rather they are assumed to exist axiomatically and are supplied with a set of basic operators whose properties are defined axiomatically. In the case of integers and reals, these operators are arithmetic operators that satisfy algebraic properties such as commutativity, associativity, distribution and simplification properties. We are currently adding support for axiomatic types and operators to the theory extension mechanism in Rodin[3]. One difference with the direct and recursive definitions is that we do not define full soundness obligations for the axiomatic definitions. For now, we assume that the theory modeller has ensured the soundness of a collection of axioms externally. We can define basic well-definedness obligations on the axioms however.

[3] `http://wiki.event-b.org/index.php/Theory_Plug-in`

Event-B already supports lambda expressions of the form $(\lambda x \cdot P(x) \mid E(x))$ where P is a constraint on x and the lambda function yields $E(x)$ for x satisfying P. Axiomatic definitions in theories also allow us to mimic other binder operators by defining operators on lambda expressions. For example, consider summation over a collection of integers that sums each expression $E(x)$ for every x satisfying predicate P:

$$\Sigma x \cdot P(x) \mid E(x)$$

This can be represented by defining a *SIGMA* operator on functions with the form

$$SIGMA(\lambda x \cdot P(x) \mid E(x))$$

SIGMA is defined as a new operator on functions satisfying the following axioms:

$$\begin{array}{rcll} SIGMA(\varnothing) & = & 0 & \\ SIGMA(\{x \mapsto y\}) & = & y & \\ SIGMA(s \cup t) & = & SIGMA(s) + SIGMA(t) & \textit{provided } s \cap t = \varnothing \end{array}$$

10 Related Work

Event-B theories are similar in principle to Isabelle [13] and PVS [15], though Isabelle and PVS theories are wider in scope. Theories in Isabelle and PVS can be used to carry significant modelling and reasoning activities. We argue that combining modelling and theory development in Event-B provides a comparable level of sophistication to that of Isabelle and PVS theories. Event-B modelling uses set theory which can provide powerful expressive power that is close to higher order logic [4]. The addition of the theory component ensures that polymorphism can be exploited to enhance the expressive power of the Event-B mathematical language.

The architecture of proof tools continues to stir up much heated debate. One of the main talking points is how to strike a reasonable balance between three important attributes of the prover: efficiency, extensibility and soundness. In [8], Harrison outlines three options to achieve prover extensibility:

1. If a new rule is considered to be useful, simply extend the basic primitives of the prover to include it.
2. Use a full programming language to specify new rules using the basic primitives. The new rules ultimately decompose to these primitives.
3. Incorporate the *reflection* principle, so that the user can add and verify new rules within the existing infrastructure.

Many theorem provers including Isabelle [13] and HOL [6] employ the LCF (Logic of Computable Functions) approach of Milner [11]. The functional language ML [12] is used to implement these systems, and acts as their meta-language. The approach taken by such systems is to use ML to define data types

corresponding to logical entities such as expressions and theorems. A number of ML functions are provided that can generate theorems; these functions implement the basic inference rules of the logic. The ML type system ensures that theorems are only constructed by the aforementioned functions. Therefore, the LCF approach offers both "reliability" and "controllability" of a low level proof checker combined with the power and flexibility of a sophisticated prover [8]. On the flip side, however, a major drawback for this approach is that each newly developed proof procedure must decompose into the basic inference rules. There are cases where this may not be possible or indeed an efficient solution, e.g., the truth table method for propositional logic [5].

The PVS [15] system follows a similar approach to LCF with more liberal support for adding external provers. This liberality comes at a risk of introducing soundness bugs. It, however, presents the user with several choices of automated provers which may ease the proving experience. A comparison between Isabelle/HOL and PVS from a user's point of view is presented in [7]. Interestingly, it mentions that "soundness bugs are hardly ever unintentionally explored" during proof, and that "most mistakes in a system to be verified are detected in the process of making a formal specification". A similar experience is reported when using the Rodin platform [10].

Schmalz [18] defines the Event-B logic using a shallow embedding in Isabelle/HOL [14]. Schmalz provides a comprehensive specification of the logic of Event-B. He gives semantics, devises soundness preserving extension methods, develops a proof calculus similar to [9], and proves its soundness. He presents a formal language for expressing rules (including non-freeness conditions) and shows how to reason about the soundness of Event-B rules. The Event-B type operators such as $\mathbb{P}$ and $\times$ are defined by means of their Isabelle/HOL counterparts. Type substitutions are central to a logic that supports polymorphism, and are also introduced. Binders, expressions and predicates are introduced and are assigned Isabelle/HOL semantics by means of a number of higher-order logic constructs. Note that Schmalz considers predicates to be HOL terms with a boolean type $\mathcal{B}$. Ways of conservatively extending the Event-B logic are outlined and the proof system of Event-B is shown to be sound [18].

11 Concluding

Polymorphic structures such as sequences, bags and stacks are very useful and common modelling elements, but they are absent from the core syntax of Event-B. Prior to our work, functions could be used to mimic operators through axiomatic definitions in Event-B contexts (e.g. see [16]) – but these functions are not polymorphic. Furthermore, from our experience of using the Rodin tool, if a new proof rule is required, a bureaucratic process has to be initiated where resources have to be allocated depending on the urgency of the request. We argue that the practical contributions of the work outlined here are that it:

– complements the Event-B methodology and make it a more rounded formalism,

- provides an appealing platform to end users because it has facilities for meta-reasoning to complement reasoning and modelling in Event-B,
- reduces the dependency on the Java programming language and specialised knowledge of Rodin architecture in order to extend the language and proof mechanisms.

Significant effort is required to develop sound theories. Theory hierarchies are a useful structuring mechanism to create operator taxonomies as is the practice in Isabelle [13]. The effort required to create and validate theories can be decomposed into two large phases:

1. Theory specification phase: new datatypes, operators and proof rules are specified. In this phase, particular attention should be paid to specifying any auxiliary operators that facilitate the use of the main newly introduced structures. In the case of the sequence theory, the *seq* operator is the main structure of the theory, and a number of auxiliary operators, e.g., $emptySeq$, $seqHead$ and $seqTail$, are also defined.
2. Theory validation phase: in this phase, proof obligations are considered and discharged by the user. This phase helps with uncovering errors in the specification of operators and proof rules, in the same way that interactive proof can reveal errors in models.

Therefore, theory development is an iterative process. It is a recurring observation that developing sound theories may take at least the same amount of effort as when developing consistent models. However, the major advantage of using theories is the reusability of definitions thanks to their polymorphic nature. Finally, the familiarity of our approach to users (reactive development, the use of proof obligations and the use of the existing Rodin user interface for specifying and validating theories) ensures that the theory component provides a practical way to extend the Event-B language and proof infrastructure.

Acknowledgements. This work was funded as part of the FP7 DEPLOY (www.deploy-project.eu) and FP7 ADVANCE (www.advance-ict.eu) projects. Various people contributed to the discussions about theory extension including Jean-Raymond Abrial, Nicolas Beauger, Andy Edmunds, Stefan Hallerstede, Alexei Iliasov, Cliff Jones, Michael Leuschel, Thomas Muller, Carine Pascal, Abdolbaghi Rezazadeh, Asieh Salehi, Matthias Schmalz, Renato Silva, Colin Snook and Laurent Voisin.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT 12(6), 447–466 (2010)

3. Abrial, J.-R.: B$^{\#}$: Toward a synthesis between Z and B. In: Bert, D., Bowen, J.P., King, S., Waldén, M. (eds.) ZB 2003. LNCS, vol. 2651, pp. 168–177. Springer, Heidelberg (2003)
4. Abrial, J.-R., Cansell, D., Laffitte, G.: "Higher-Order" Mathematics in B. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002. LNCS, vol. 2272, pp. 370–393. Springer, Heidelberg (2002)
5. Di Cultura, I.T., Armando, A., Armando, A., Cimatti, A., Cimatti, A.: Building and executing proof strategies in a formal metatheory. In: Torasso, P. (ed.) AI*IA 1993. LNCS, vol. 728, pp. 11–22. Springer, Heidelberg (1993)
6. Gordon, M.: HOL: A Machine Oriented Formulation of Higher Order Logic (1985)
7. Griffioen, D., Huisman, M.: A comparison of PVS and isabelle/HOL. In: Grundy, J., Newey, M. (eds.) TPHOLs 1998. LNCS, vol. 1479, pp. 123–142. Springer, Heidelberg (1998)
8. Harrison, J.: Metatheory and Reflection in Theorem Proving: A Survey and Critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK (1995) `http://www.cl.cam.ac.uk/~jrh13/papers/reflect.dvi.gz`
9. Mehta, F.: A Practical Approach to Partiality – A Proof Based Approach. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 238–257. Springer, Heidelberg (2008)
10. Mehta, F.: Proofs for the Working Engineer. PhD Thesis, ETH Zurich (2008)
11. Milner, R.: Logic for computable functions: description of a machine implementation. Technical report, Stanford, CA, USA (1972)
12. Milner, R., Tofte, M., Harper, R.: The definition of Standard ML. MIT Press, Cambridge (1990)
13. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle Logics: HOL (2000)
14. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
15. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Language Reference (2001)
16. Robinson, K.: Reconciling axiomatic and model-based specifications reprised. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 223–236. Springer, Heidelberg (2008)
17. Romanovsky, A., Thomas, M. (eds.): Industrial Deployment of System Engineering Methods. Springer (2013)
18. Schmalz, M.: The Logic of Event-B. Technical Report 698, ETH Zurich, Switzerland (2010), `http://www.inf.ethz.ch/research/disstechreps/techreports`

Simulink Timed Models for Program Verification

Ana Cavalcanti[1], Alexandre Mota[2], and Jim Woodcock[1]

[1] Department of Computer Science, University of York,
York, YO10 5DD, England
[2] Centro de Informática, Universidade Federal de Pernambuco, Brazil

Abstract. Simulink is widely used by engineers to provide graphical specifications of control laws; its frequent use to specify safety-critical systems has motivated work on formal modelling and analysis of Simulink diagrams. The work that we present here is complementary: it targets verification of implementations by providing a refinement-based model. We use *CircusTime*, a timed version of the *Circus* notation that combines Z, CSP, and Morgan's refinement calculus with a time model, and which is firmly based on Hoare & He's Unifying Theories of Programming. We present a modelling approach that formalises the simulation time model that is routinely used for analysis. It is distinctive in that we use a refinement-based notation and capture functionality, concurrency, and time. The models produced in this way, however, are not useful for program verification, due to an idealised simulation time model; therefore, we describe how such models can be used to construct more realistic models. This novel modelling approach caters for assumptions about the programming environment, and clearly establishes the relationship between the simulation and implementation models.

Keywords: Simulink, Z, CSP, *Circus*, time, refinement, modelling.

On the occasion of He Jifeng's 70th birthday.

1 Introduction

The use of Simulink diagrams [17] for the specification of control laws is pervasive in industry. Various approaches enrich the current Simulink facilities for analysis of diagrams with techniques based on formal methods [15,2,6]. Denney & Fischer [7], for example, propose the use of the AutoCert verification system to construct a natural language report explaining where code uses specified assumptions and why and how it complies with requirements (though, significantly, not for timing aspects). In contrast, our work recognises the need to verify implementations. Automatic code generation does not usually provide enough assurance: even when generators are reliable, restrictions on performance or resources often require changes to the code.

In previous work [4,3], we covered functional and behavioural aspects of diagrams and implementations. We cater for the inherent parallelism in diagrams and the verification of complete programs, including their scheduling components. For modelling and reasoning, we use *Circus* [20], a flexible integration of

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 82–99, 2013.

Z [30], CSP [23], and Morgan's refinement calculus [18], with formal foundations underpinned by Hoare & He's Unifying Theories of Programming (UTP) [13]. *Circus* is a mature notation with a sound semantics implemented in tools: it was first introduced in 2001 in [28], given a formal semantics in 2002 [29], and subsequently mechanised in ProofPowerZ, a HOL-based theorem prover, in [20,21], and in Isabelle/HOL [8,9].

We generate formal models automatically, and apply a refinement tactic in ProofPowerZ to prove that the model of the program conforms to (refines) the model of the diagram. Automation is enabled by knowledge of the structure of the automatically generated models, and of the correspondence between diagram and program components.

What we have not covered before is the time model embedded in the diagrams. In [4,3], we use synchronisation to model the cycles of the diagram, which are in fact defined by simulation time parameters. In this approach, we cannot cater for multi-rate diagrams and, most importantly, have to consider partial program models that do not capture the use of timing primitives (like delay commands). To produce a model of a program, we consider a slice that removes all variables related to time control; this can potentially mask an error.

In this paper we present a novel modelling approach to cover the time properties of diagrams. Our approach uses *CircusTime* [27], a timed version of *Circus* with both timeout and deadline operators. *CircusTime* was first introduced in 2002 in [25], and given a complete formal semantics in UTP in [26,27].

In our new approach, we capture the idealised-time model adopted in the Simulink simulator as well as its data-flow model, which embeds some calculations (functional properties) and concurrent behaviour. Since we are interested in software implementations, we consider only diagrams with a discrete-time model; but we can cover multi-rate diagrams as well.

The idealised-time model of the simulation is not implementable, since it involves infinitely fast computations. So we also provide a realistic model used by typical implementations that run on real-time computers. This programming model embeds assumptions about the environment; in particular, we consider the assumptions adopted in the standard Simulink code generator, but our approach can be adapted for different real-time computers. The timed programming model is the appropriate starting point for the verification of programs.

The programming model is written in terms of the simulation model, so that we formalise the way in which the assumptions made about the programming environment affect the simulation model. Engineers use the simulation model in the analysis and validation of diagrams and corresponding control laws, so it is important to understand the way in which it is reflected in programs. Additionally, the *CircusTime* model that captures the Simulink idealised-time model is in direct correspondence with the informal description of the simulator. Its use to define the programming model increases our confidence in its validity.

With the use of *CircusTime*, we can provide very faithful models of the diagram and of the assumptions about the environment; it is also possible to model

programs in a direct way. All this reduces the risk of introducing modelling errors that compromise verification.

Ultimately, the simulation time model is defined by the solver, a component of the simulator that determines the simulation steps. For simplicity, we consider a fixed-step solver, where the step size is constant; this is the solver used to generate code for a real-time computer. It is not difficult to generalise our model for a variable-step solver. In this case, the step size changes with time, so that steps that do not present any changes to the output are omitted.

In the next section we give a brief overview of Simulink diagrams, *Circus*, and *CircusTime*. In Section 3, we present our approach to construct timed simulation models. The programming models are discussed in Section 4. Finally, in Section 5, we draw our conclusions and discuss related and future work.

2 Simulink, *Circus*, and *CircusTime*

This section describes the modelling notations used in our work.

2.1 Simulink

A control law diagram is a graph whose nodes are blocks that embed an input, an output, or some computation, and whose edges are wires that connect the input and output ports of the blocks to define data flow. The behavioural model embedded in a diagram is a cyclic execution, where in each iteration inputs are read and outputs are calculated and produced.

At the top of Figure 1, we present the Simulink diagram for a PID controller. (This is the same example used in [4,3].) The rounded boxes are input and output blocks that represent the inputs and outputs of the system. Inputs are represented by outputs of input blocks, which work in the same way as any other block. Outputs are the inputs of the output blocks. In Figure 1, we have inputs E, Kp, Ki, and Kd, and output Y. The rectangles and circles are blocks that define particular calculations. The Sp block, for instance, is a simple multiplication.

Subsystem blocks are defined by other diagrams. In Figure 1, the blocks Diff and Int are defined by the diagrams at the bottom of Figure 1. A control law for a real system may reach hundreds of pages of hierarchical diagrams.

Blocks can have state. A Unit Delay, used in the diagrams in Figure 1, for instance, records in its state the value of its last input.

A block also has a sample time, characterised by a sampling period and an offset, which indicates when, during simulation, it produces outputs and updates its state, if any. Additionally, each port can have a different sample time. If the blocks do not all have the same sample time, we have a multi-rate diagram.

A simulation is described by a series of steps; a solver determines the time of the next simulation step. We assume that the default step size and offset determined by the fixed-step solver are used, since they guarantee that all sample times are covered. The default step size is the greatest common divisor of all sample times. A block's state is updated and a new output is generated only when the time of a step is a multiple of the sample time of the block.

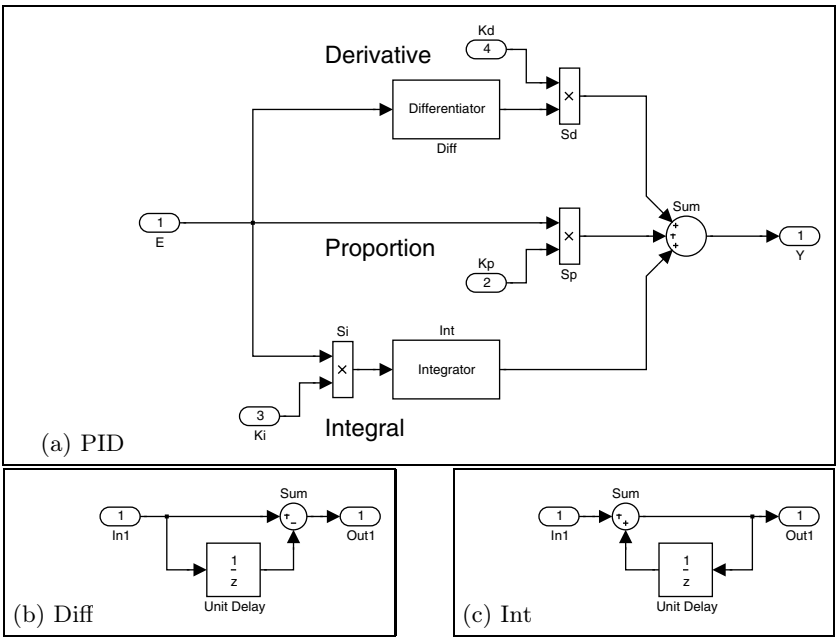


Fig. 1. PID (Proportional Integral Derivative) controller

In Section 3, we formalise this simulation time model using our novel approach based on *CircusTime*, which we describe next.

2.2 *Circus* and *CircusTime*

A *Circus* model is a sequence of definitions: Z paragraphs, channel declarations, and process definitions. Several examples are presented in the next section (see Figures 5, 6, 8, 2, and 3). After a simple example in this section, we explain details of the notation as it is used.

$$PIDTSpec \mathrel{\widehat=} \left(\begin{array}{l} \textbf{\textit{Wait}}\ 1 \fatsemi (PID[E, Kp, Ki, Kd, Y := Ed, Kpd, Kid, Kdd, Yd]) \setminus \{\!|step|\!\} \\ \quad [\![Internal]\!] \\ Interface(1) \end{array} \right) \setminus Internal$$

Fig. 2. Programming model of the PID

Just like in CSP, systems and their components are described by processes that communicate with each other and the environment using channels. In *Circus* and *CircusTime*, however, a process encapsulates some state (defined just like in Z). More precisely, in an explicit process definition, we define state components and invariants using a Z schema and define behaviour using actions. To specify an action, we use a mixture of CSP constructs, Z data operations, and guarded

$$
\begin{array}{l}
\textbf{process}\ Interface \mathrel{\widehat=} stepSize : \mathbb{R} \bullet \textbf{begin} \\
\quad \textbf{state}\ IS \mathrel{\widehat=} [\, Ev, Kpv, Kiv, Kdv, Et, Kpt, Kit, Kdt, Yv : \mathbb{R} \,] \\
\quad EInp \mathrel{\widehat=} \left(\begin{array}{l} (\sqcap\, d : 0 \,..\, stepSize \bullet \textbf{Wait}\ d \,;\, (E?x \to Ev, Et := x, d)\ \textbf{deadline}\ 0) \,; \\ (\mu\, X \bullet E?x \to Ev := x \,;\, X) \overset{stepSize - Et}{\triangleright} Skip \end{array} \right) \\
\quad \ldots \\
\quad Input \mathrel{\widehat=} EInp \mid\mid\mid KpInp \mid\mid\mid KiInp \mid\mid\mid KdInp \\
\quad YOut \mathrel{\widehat=} (\sqcap\, d : 0 \,..\, stepSize \bullet \textbf{Wait}\ d \,;\, (\bigsqcup v : \mathbb{R} \bullet Y!v \to Yv := v)\ \textbf{deadline}\ 0) \\
\quad Output \mathrel{\widehat=} YOut \\
\quad InputD \mathrel{\widehat=} Ed!Ev \to Skip \mid\mid\mid Kpd!Kpv \to Skip \mid\mid\mid Kid!Kiv \to Skip \mid\mid\mid Kdd!Kdv \to Skip \\
\quad OutputD \mathrel{\widehat=} Yd?x \to \{\, Yv = x \,\} \\
\quad \bullet\ (\mu\, X \bullet (Input \mid\mid\mid Output) \,;\, (InputD \mid\mid\mid OutputD) \,;\, X) \\
\textbf{end}
\end{array}
$$

Fig. 3. *Interface* process for the PID

commands. In the case of *CircusTime*, we additionally use wait, timeout, and deadline operators in the style of Timed CSP [22].

To compose processes, we use CSP operators (for parallelism, choice, and hiding, for instance). Here, we use the alphabetised parallel operator [23], where the set of channels used by each process is defined and they are required to synchronise on the intersection of these sets. We also use hiding, which, just like in CSP, removes a channel, or set of channels, from the interface of a process.

The semantic model for *Circus* has been chosen to support a simple and intuitive notion of refinement: one process Q refines another P, providing that every behaviour of Q is also a behaviour of P. In this way, if specification P is refined by implementation Q, then there is nothing that Q could do that would be forbidden by P, and all its behaviours are specified behaviours. The semantics of *Circus* is a timed extension of the failures-divergences semantics of CSP [12] in the spirit of [22], and so the notion of refinement includes subtle testing of nondeterminism and timing properties.

To illustrate *CircusTime*, we use a timed version of a small example that first appeared in [28], where we used the untimed version of *Circus* to specify a Fibonacci series generator. This is a simple process that generates the Fibonacci series on a channel named "*out*"; the process is described in full in Figure 4. *Circus* processes have encapsulated state that is defined in Z; in the process *Fib*, the state is defined using the schema named *FibState*, which introduces two natural numbers, x and y.

The state is initialised in *InitFibState* to give both state components the initial value of 1. *InitFib* is an action, defined using CSP with embedded references to the state defined in Z. This action first initialises the state using *InitFibState*, then it outputs the first two numbers in the Fibonacci series. It does this with a deadline of 0, which makes the outputs occur instantaneously; after each output, the action pauses for one time interval.

The main work in generating the series is done in the action *OutFib*. First, we define *OutFibState*, which updates the state components: this operation changes the state ($\Delta FibState$) and has one output, defined in the Z convention as $next!$. The effect of this state operation is to set the output to be the same as y', the newest member of the series, which is merely the sum $x + y$, and to copy the value of y to x'.

OutFib itself updates the state using *OutFibState* and then outputs the value of *next* punctually, as before, and it does this repeatedly: the fixed-point operator "μ" introduces tail-recursive iteration. A local-variable block scopes the value of *next*.

After all these schema and action definitions, the real business begins: the main behaviour of the process *Fib* is defined as the composition $InitFib;\ OutFib$.

$$
\begin{array}{l}
\textbf{process}\, Fib\ \widehat{=} \\
\quad \textbf{begin} \\
\qquad \textbf{state}\ FibState == [\, x, y : \mathbb{N}\,] \\
\qquad InitFibState == [\, FibState' \mid x' = y' = 1\,] \\
\qquad InitFib\ \widehat{=} \\
\qquad\quad InitFibState; \\
\qquad\quad (out!1 \rightarrow Skip)\ \textbf{deadline}\ 0;\ \textbf{Wait}\ 1; \\
\qquad\quad (out!1 \rightarrow Skip)\ \textbf{deadline}\ 0;\ \textbf{Wait}\ 1 \\
\qquad OutFibState == [\, \Delta FibState;\ next! : \mathbb{N} \mid next! = y' = x + y \land x' = y\,] \\
\qquad OutFib\ \widehat{=} \\
\qquad\quad \mu\, X \bullet \\
\qquad\qquad \textbf{var}\ next : \mathbb{N} \bullet \\
\qquad\qquad\quad OutFibState;\ (out!next \rightarrow Skip)\ \textbf{deadline}\ 0;\ \textbf{Wait}\ 1;\ X \\
\quad \bullet\ InitFib;\ OutFib \\
\quad \textbf{end}
\end{array}
$$

Fig. 4. A timed Fibonacci series generator

3 Simulation Models

In this section, we propose a novel approach to construct *CircusTime* simulation models of Simulink diagrams. Like the *Circus*-based strategy, it can be used to generate models automatically [31]. As already said, it produces richer models that cater also for the timing aspects of a larger set of diagrams. We have more faithful models, which we have demonstrated to be in direct correspondence with the simulator behaviour, and as a side effect we also get more compact models.

Our input is a diagram compiled using the fixed-step discrete solver. This means that there is no connection between blocks with incompatible sample times and the discrete sample time of all blocks used for simulation has been defined. (It is possible to leave the sample time of a block to be determined from the context; compilation determines all values.) This is the approach taken, for example, by the MATLAB code generator (Real-time Workshop).

The output of our modelling strategy is a *CircusTime* specification. Its first paragraphs declare channels. Inputs and outputs of the diagram and of the blocks are represented by channels. For the PID, we have the following declaration.

$$\textbf{channel}\ E, Kp, Ki, Kd, Y, Si\_out, Diff\_out, Int\_out, Sd\_out, Sp\_out : \mathbb{U}$$

Basically, the channels that represent the inputs and outputs of the diagram are named after the corresponding blocks. The internal wires are represented by channels named after the block that has an output port connected to it. For instance, the wire that connects Diff to Sd in Figure 1 is represented by a channel $Diff\_out$. (As explained in [31], a few special cases need to be considered in the naming rules, but for the purpose of the discussion here, this view is enough.)

The blocks, the solver, and the diagram itself, are modelled by processes. The solver process synchronises with the block processes on a channel *step*. It is used to indicate the occurrence of a simulation step.

$$\textbf{channel}\ step : \mathbb{R}$$

We have a discrete-time model, but the sample times and offset can be real numbers, so the type of *step* is $\mathbb{R}$. It is available in the HOL-based theorem prover ProofPower-Z.

The third paragraph of the model declares a type $SampleTime$.

$$SampleTime == [\, sP, o : \mathbb{R} \,]$$

It contains records whose components sP and o define a sample period and an offset. Each block process has a constant of this type to represent its sample time. Below, we explain how the diagram, block, and solver processes are defined.

3.1 Diagram

The processes that model the blocks and the process that models the solver are all composed in parallel to define the process that models the diagram. For our example, this process is sketched below; its name is that of the diagram.

$$\textbf{process}\ PID \mathrel{\widehat=} \left(\begin{array}{l} Si \ \{\!|\, E, Ki, Si\_out, step \,|\!\} \\ \quad \| \\ Diff \ \{\!|\, E, Diff\_out, step \,|\!\} \\ \quad \| \qquad \ldots \| \\ FixedStepDiscreteSolver(1,0) \ \{\!|\, step \,|\!\} \end{array} \right) \setminus \{\!|\, Si\_out, Diff\_out, \ldots \,|\!\}$$

The alphabet of each process that models a block includes the channels that are used to represent its inputs and outputs, besides *step*. This reflects the fact that the behaviour of each block is independent, but the way in which their inputs and outputs are connected defines a data flow. In the model, synchronisation on the shared channels between block processes establishes data flow.

process $Diff \mathrel{\widehat=}$ **begin**

$st : SampleTime$

$st.sP = 1 \land st.o = 0$

state $Diff\_State == [\, pid\_\_Diff\_\_UnitDelay\_state : \mathbb{R};\ ;\ Out1 : \mathbb{R}\,]$

Init

$pid\_\_Diff\_State'$

$pid\_\_Diff\_\_UnitDelay\_state' = 0$

$Calculate\_pid\_\_Diff == \ldots$

$$\bullet \left(\left(\begin{array}{l} Init\,; \\ \mu\, X \bullet \\ \left(\begin{array}{l} step?cT \rightarrow \\ \left(\begin{array}{l} \textbf{var}\ In1 : \mathbb{U} \bullet \\ \quad E\,?x \rightarrow In1 := x\,; \\ \quad \textbf{if}\ (cT - st.o \geq 0) \land ((cT - st.o) \bmod st.sP = 0) \rightarrow \\ \qquad Calculate\_pid\_\_Diff \\ \quad [\!] \ (cT - st.o < 0) \lor ((cT - st.o) \bmod st.sP \neq 0) \rightarrow Skip \\ \quad \textbf{fi}\,; \\ \quad Diff\_out\,!Out1 \rightarrow Skip \end{array} \right) \\ \textbf{deadline}\ 0 \end{array} \right) \end{array} \right) ; X \right)$$

end

Fig. 5. *CircusTime* model of the block Diff

The process *FixedStepDiscreteSolver* models the solver; it takes the step size and offset of the diagram as parameters. In our simple example, all blocks have sampling period 1 and offset 0, so the solver uses step size 1 and offset 0.

The channels that represent internal wires (in our example, *Si_out*, *Diff_out*, and so on) are hidden. In this way, the channels in the interface of the diagram process are only those that represent inputs and outputs of the system, and *step*.

3.2 The Blocks

A block process is defined explicitly, independently of whether the block is simple, like Sd in our example, or a subsystem, like Diff. We consider here blocks with a single sample time; port-based sample time is addressed in Section 3.4.

Figure 5 sketches the model for Diff, a process named *Diff*. An explicit process definition is composed of a sequence of paragraphs. In a block process, we first declare a constant *st* of type *SampleTime* and define the value of its fields. This is defined in the diagram: although it does not (necessarily) appear in its graphical representation, it appears in its textual representation produced by Simulink.

A distinguished paragraph declares the state of the process using a schema named after it; in our example, *Diff_State*. The state components record the

state of the block, and the last calculated output value(s). In our example, we have $pid\_\_Diff\_\_UnitDelay\_state$, which records the state of the Unit Delay block used in the Diff diagram, and $Out1$, corresponding to the single output of Diff.

A schema *Init* defines the state initialisation in the standard Z way. A declaration like $pid\_\_Diff\_State'$ introduces dashed versions of the state components to represent their values after initialisation. The components that represent the block state are initialised as determined in the block definition (included in the textual representation of the diagram). In our example, the initial value of the Unit Delay state, represented by $pid\_\_Diff\_State'$, is 0. The components that correspond to block outputs, like $Out1$ in our example, are not initialised.

The action at the end of a process definition (after the $\bullet$) is the main action that specifies its behaviour. In a block process, we have a call to *Init*, followed by a recursion (introduced by the μ operator). Each of its iterations models a simulation step. It starts with a communication $step?cT$ on the channel *step* to input the current time cT, followed by an interleaving of the inputs, which are recorded in local variables $In1$, $In2$, and so on. In our example, we have just one input $E?x$, and the associated assignment of the input value x to $In1$.

A conditional compares the current and sampling times. If the block offset is over $(cT - st.o \geq 0)$ and we have a sample time hit $((cT - st.o) \bmod st.sP = 0)$, we calculate the outputs and update the state. Otherwise, nothing happens; the *Skip* action terminates immediately without changing the state.

The required calculations and updates are determined by the functionality of the block (or its diagram, in the case of a subsystem block like Diff). We rely on an industrial tool, namely ClawZ [1], to produce a Z specification for that.[1] (This is the same approach that we take in [3,4].) ClawZ deals with sequential behaviour; we have extended this to deal with concurrency and timing aspects. For each block, ClawZ produces a Z schema, which we use to define the *Circus Time* process. In our example, we have the $Calculate\_pid\_\_Diff$ schema, whose definition is constructed using ClawZ. We omit it here, since these details are not relevant for our discussion. In the main action of *Diff*, when there is a sample time hit, we call $Calculate\_pid\_\_Diff$.

In any case, the (last calculated) outputs are communicated in interleaving. These are either the outputs that have just been calculated, or those calculated in the previous sample time hit. In our example, we have a single output: we communicate $Out1$ through the channel $Diff\_out$.

All this is carried out instantaneously, that is, with **deadline** 0. This captures the idealised-time model of the simulation, where the system is quiescent between the simulation steps, but all the inputs, calculations, updates, and outputs are performed instantaneously (and infinitely fast), when there is a time hit. This is, of course, not an approach that can be taken by a program.

As explained previously, the state components that correspond to an output, like $Out1$, are not initialised. If the solver takes a simulation step before the first

[1] See `www.lemma-one.com/clawz_docs/` for more information about the ClawZ tool, including a user guide to the tool with a simple complete worked example of a Simulink model file, its corresponding Ada code, and a proof of correctness.

sample time hit of the block, the value of the output is arbitrary. For Simulink diagrams whose simulation does not generate any errors, however, such a situation does not arise. Here, as also already said, we are only concerned with compiled diagrams that do not produce simulation errors.

In Section 4, we discuss how the model presented here can be used to construct a model compatible with the restrictions of a real-time computer.

3.3 Solver

To ensure correct timing, the solver process uses the channel *step* to communicate the current time to all blocks in each simulation step. The model of the solver embeds a clock that is indirectly used by all the blocks.

The model of the solver is the same for all diagrams; it is presented in Figure 6. The step size sS and offset o are taken as parameters, which are instantiated appropriately for each diagram. The state, defined by the schema *Clock* contains a single component cT to record the current time.

$$\textbf{process}\ \mathit{FixedStepDiscreteSolver} \mathrel{\widehat=} sS, o : \mathbb{R} \bullet \textbf{begin}$$

$$\textbf{state}\ \mathit{Clock} == [\, cT : \mathbb{R} \,]$$

$$\bullet \left(\begin{array}{l} cT := o \,;\ \textbf{Wait}\ o \,; \\ (\mu\, X \bullet (\mathit{step}!cT \rightarrow \mathit{Skip})\ \textbf{deadline}\ 0 \,;\ cT := cT + sS \,;\ \textbf{Wait}\ sS \,;\ X) \end{array} \right)$$

$$\textbf{end}$$

Fig. 6. *CircusTime* model of a fixed-step discrete solver

In the main action, cT is initialised to o, since nothing happens in the simulation before the diagram offset time. Next, after a wait period of o time units, there is an iteration corresponding to simulation steps. In each iteration, an instantaneous communication (with deadline 0) over the channel *step* outputs the current time cT. Afterwards, cT is increased by the step size sS, and there is a waiting period of sS time units.

3.4 Multi-rate Diagrams

Our modelling approach caters for multi-rate diagrams. As already explained, if the blocks have different sample times, the step size and offset of the solver guarantee that they are covered. In the model, at each simulation step, all block processes read inputs and produce outputs, but output calculations and state updates occur only when there is a hit. For blocks with port-based sample times, however, we need to define the block processes differently.

Port-based sample times occur in rate-transition blocks, which have one input and one output port with different sample times, and custom blocks defined by programs. In what follows, we explain our treatment of rate-transition blocks. Custom blocks can be handled in a similar way, but the behaviour when there is a sample-time hit is defined programmatically. Modelling such blocks requires modelling, for example, a C program instead of using a ClawZ schema.

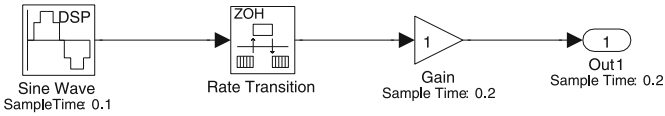


Fig. 7. Multi-rate diagram: rate transition block

$$
\begin{array}{l}
\textbf{\textit{process}}\ Rate\_Transition \mathrel{\widehat{=}} \textbf{\textit{begin}} \\
\quad st_1, st_2 : SampleTime \\
\quad \hline
\quad st_1.sP = 0.1 \land st_1.o = 0 \land st_2.sP = 0.2 \land st_1.o = 0 \\
\quad \textbf{\textit{state}}\ rt\_\_Rate\_Transition\_State = [\, In1, Out1 : \mathbb{U} \,] \\
\quad \ldots \\
\quad \bullet \left(\begin{array}{l}
\mu\, X \bullet \\
\left(\begin{array}{l}
step?cT \rightarrow \\
\left(\begin{array}{l}
Sine\_Wave\_out?x \rightarrow \\
\left(\begin{array}{l}
\textbf{\textit{if}}\ (cT - st_1.o \geq 0) \land ((cT - st_1.o) \bmod st_1.sP = 0) \rightarrow \\
\qquad In1 := x \\
[\!]\ (cT - st_1.o < 0) \lor ((cT - st_1.o) \bmod st_1.sP \neq 0) \rightarrow \\
\qquad Skip \\
\textbf{\textit{fi}}
\end{array} \right) ; \\
\left(\begin{array}{l}
\textbf{\textit{if}}\ (cT - st_2.o \geq 0) \land ((cT - st_2.o) \bmod st_2.sP = 0) \rightarrow \\
\qquad rt\_Rate\_Transition \\
[\!]\ (cT - st_2.o < 0) \lor ((cT - st_2.o) \bmod st_2.sP \neq 0) \rightarrow Skip \\
\textbf{\textit{fi}}
\end{array} \right) ; \\
One\_out\,!Out1 \rightarrow Skip
\end{array} \right) \\
\textbf{deadline}\ 0 \\
X
\end{array} \right) ;
\end{array} \right)
\end{array}
$$

end

Fig. 8. *CircusTime* model of a rate-transition block: zero-order hold

For rate-transition blocks, we need to consider whether its input port is slower (has a longer sampling period) or faster than the output port, as this determines its behaviour when there is a hit. If the input is faster, then the behaviour is that of a zero holder: the block holds its input until there is a hit for the output port. If the input is slower, then the behaviour is that of a unit delay: it outputs the input from a previous hit.

A diagram involving a rate-transition block is provided in Figure 7. In this example, all offsets are 0, but the period of the input port of the rate-transition block (and of the Sine-Wave block) is 0.2, and that of the output port (and of the Gain and Out blocks) is 0.1. The diagram step size is therefore 0.1.

The construction of the model of this diagram can proceed much as before, the only difference concerning the rate-transition block process, which is described in Figure 8. First, it records two sample times st_1 and st_2, one for each of its ports.

Just as before, in each simulation step, the input is taken and the output is produced. Due to the lack of synchrony between inputs and outputs, we do not

$$
\left(\begin{array}{l}
Init\,; \\
\left(\begin{array}{l}
\mu\, X \bullet \\
\left(\begin{array}{l}
step?cT \rightarrow \\
\left(\begin{array}{l}
Sine\_Wave\_out?x \rightarrow \\
\left(\begin{array}{l}
\textbf{if}\ (cT - st_1.o \geq 0) \land ((cT - st_1.o)\ \mathsf{mod}\ st_1.sP = 0) \rightarrow \\
\qquad rt\_Rate\_Transition \\
[\!]\ (cT - st_1.o < 0) \lor ((cT - st_1.o)\ \mathsf{mod}\ st_1.sP \neq 0) \rightarrow \\
\qquad Skip \\
\textbf{fi}
\end{array}\right) \\
One\_out\,!\,Out1 \rightarrow Skip
\end{array}\right)\,; \\
\textbf{deadline}\ 0
\end{array}\right)\,; \\
X
\end{array}\right)
$$

Fig. 9. *CircusTime* model of a rate-transition block: unit delay (main action)

keep only the most recently output value from one step to the next, but also the most recent input. We have both $In1$ and $Out1$ as state components.

If a simulation step is a hit for the sample time of the input, $In1$ is updated, otherwise the input taken is ignored. If there is a hit for the sample time of the output, then it is calculated. In the case of a zero-order hold block, the calculation, as defined by ClawZ, updates $Out1$ to the value of the most recent input $In1$. This is defined in the (omitted) schema $rt\_Rate\_Transition$.

For a rate-transition block with a slower input we have a block process whose main action is shown in Figure 9. In this case, we have an additional state component *state* as indicated by ClawZ, and the *Init* action initialises the *state* as defined in the block properties, and also captured by ClawZ.

The output calculations and state updates are determined by the sample-time hit of the slow port. Here, it is the input, so we do not need to check for hits of the output nor record the inputs in the state. The $rt\_Rate\_Transition$ schema defines that $Out1$ is assigned the current value of the state, which becomes the freshly input value x. This is the standard definition of a unit delay.

In the following section, we describe how these simulation models can be used to define models appropriate for program verification.

4 Programming Models

The idealised simulation model requires the calculations and communications to take place infinitely fast. For program verification, we need a model that captures the assumptions that allow us to conclude that an implementation is correct, from the timing as well as the functional point of view, even though it is restricted by the performance of the real-time computer on which it runs.

All calculations and communications take place instantaneously at each simulation time step. For programs, we expect a time line where all calculations and communications take place during the intervals defined by the hits. This is, for example, the view adopted by the MATLAB code generator. The (implicit)

assumption adopted in the default configuration of this code generator is that the simulation steps define execution cycles. Additionally, the environment keeps the inputs constant and available during each cycle, and is ready to accept an output at any point during the cycle.

Here, we explain how the simulation model can be used to construct a model for program verification. The result is a new *CircusTime* process; for our PID example, this is *PIDTSpec*, presented in Figure 2. Roughly, it is defined by composing the simulation model of the diagram, in our case, *PID*, in parallel with a process *Interface* that handles the inputs and outputs of the system to capture the assumptions about the environment.

We need to adapt the simulation model of the diagram in three ways. First, we need to address the fact that the simulation provides outputs already at the first hit, even if it is at time 0, while the program needs to take some time before it can produce results. In the program verification model, therefore, the simulation model is used after a wait period. In this way, the simulation time line is shifted, and during that initial period the computations can start. We use the step size as the wait period; in our example, this is 1. The assumption is that, for all blocks, one step is enough to calculate the outputs and make the state updates. This is again the view taken by the MATLAB code generator.

A second, most important observation is that the inputs and outputs of the simulation model correspond to those of the system. It is, however, the *Interface* process that needs to handle these communications. For this reason, we use the simulation diagram process obtained by renaming the input and output channels. The new channels are used for internal communication with *Interface*. For the *PID*, we declare the channels Ed, Kpd, Kid, Kdd, and Yd, and in the definition of *PIDTSpec*, we use $PID[E, Kp, Ki, Kd, Y := Ed, Kpd, Kid, Kdd, Yd]$. We also define a channel set *Internal* to include all the new channels, which are hidden in the programming model (see Figure 2).

A final observation is that the *step* channel is used to mark the simulation steps, and has no role in the program. So, it is hidden as well.

Figure 3 sketches the definition of the *Interface* process used in the specification of *PIDTSpec*. In all cases, *Interface* takes the diagram step size as a parameter. If the offset of the diagram is not 0, then it is preceded by a corresponding wait in the programming model. For the PID, this is not necessary.

The state of *Interface* includes two components for each input of the diagram, and one for each output. For an input E, for instance, the component Ev records the last value input, and Et records the time the input was first read in the current cycle. The output components hold the values output by the program. For the PID, the inputs give rise to state components Ev, Kpv, Kiv, and Kdv, and Et, Kpt, Kit and Kdt, and the output to a component Yv.

The behaviour of *Interface* is characterised by iterations that correspond to the simulation steps of the diagram. During each of them, *Interface* interacts with the environment; it reads the inputs one or more times, and produces the outputs as required. At the end of each step, it interacts with the simulation process to provide its inputs and take its outputs.

Correspondingly, in the specification of *Interface*, the main action's iterations correspond to the simulation steps. A sequence (;) splits each iteration into two parts, corresponding to the period before the simulation step and to its end, which is the exact moment of a simulation time step. During each iteration, the behaviour is defined by the interleaving ($|||$) of actions *Input* and *Output*, which interact with the environment. At the simulation time step, the behaviour is given by the interleaving of *InputD* and *OutputD*, which interact with the simulation diagram model.

As detailed later on in this section, the values of the state components Ov that are output to the environment in *Output* are chosen angelically. In our example, we have a single component Yv corresponding to an output, and its value is chosen angelically. An angelic choice is resolved in a way that ensures, if at all possible, that the program does not abort. In programming terms, it provides a backtracking mechanism. In *OutputD*, the value x provided by the simulation model for the output is compared to that of the corresponding state component Ov in an assumption $\{Ov = x\}$. In our example, we have $\{Yv = x\}$. An assumption $\{Ov = x\}$ is an action that aborts if Ov is different from x, but otherwise skips. Since the value of Ov is angelically chosen, the assumption is guaranteed to hold; effectively, it forces the value Ov to be chosen correctly.

Angelic nondeterminism is typically used as a specification construct. This is certainly the case here. Refinement of our models to feasible programs leads to implementations that make the appropriate calculations to determine the value to be output. Use of backtracking is not really practical or necessary.

Since we have an assumption that the values of the inputs are constant during a cycle, each input can be read any number of times during each iteration, but at least once. For each input, we define an action that specifies this behaviour.

For the input E of the PID model, for instance, we have the action *EInp*. The internal choice ($\sqcap$) over a delay d allows the first input to happen at any time during the iteration. More precisely, the wait of d time units followed by the instantaneous communication over E specifies that the input occurs exactly after d time units. Additionally, the internal choice of d in the specification model means that a program can choose a value for d freely: it can carry out the input at any time, whenever needed, during the iteration. In the specification model, that time is recorded in the state component Et. The value input itself is recorded in Ev. After that first communication, additional inputs on E can happen any number of times, during the rest of the iteration. After $stepSize - Et$ time units, however, the iteration finishes, and so does the input action. A timeout (operator $\overset{d}{\triangleright}$) interrupts its recursive execution in favour of *Skip*.

The inputs are all independent, so the action *Input* that specifies the program inputting behaviour is defined by the interleaving ($|||$) of the actions that handle each of the diagram inputs. In Figure 3, we omit the definitions of *KpInp*, *KiInp*, and *KdInp*, which are similar to that of *EInp*.

Each output is produced just once, but at any time, during the iteration. For each output, we have an action in *Interface*. In our example, we have just *YOut*, because we have only one output. Like in an input action such as *EInp*, in an

output action we use an internal choice to leave open the choice of when the output is produced. As already mentioned, an angelic choice ($\sqcup$) determines the value v to be output and recorded in the corresponding state component.

If there are several outputs, the *Output* action is defined as their interleaving. In the case of the PID, we have just one output, so *Output* is just *YOut*.

In the interaction between *Interface* and the simulation process, the inputs are produced in interleaving; this is defined by the action *InputD*. Similarly, *OutputD* takes all outputs in interleaving. The values received from the simulation model, however, are compared to those previously output. In Figure 3, after reading the value x through the internal channel Yd, we have the assumption $\{ Yv = x \}$. As explained above, it determines the angelic choice in *Output*.

The action *OutputD* interleaves all communications to receive outputs from the simulation process and their associated assumptions. For the PID, which has just one output, no interleaving is needed. The deadlines in the simulation model guarantee that all communications in *InputD* and *OutputD* are instantaneous.

The step size of the simulation diagram process, as defined in the instantiation of the solver process, and that of the *Interface* process should be the same. This can be easily ensured when the models are generated automatically.

The external channels of *PID* and *PIDTSpec* are the same. This holds in general for simulation and programming models constructed as described here.

5 Conclusions

In this paper, we propose a modelling strategy to use *CircusTime* to capture both timing properties embedded in the simulation model of a Simulink diagram and the timing assumptions embedded in a typical programming environment. The models produced capture functional, behavioural, and timing aspects of diagrams. The use of a refinement notation, and the consideration of programming concerns, make the models useful for program verification.

The simulation model is in direct correspondence with the description of the Simulink simulator. Using *CircusTime* and angelic nondeterminism, we construct a programming model that records the environment assumptions, but uses the simulation model to specify functionality and data flow. We overcome three challenges in establishing the connection between the two models: in the simulation model, outputs can be produced immediately at time 0, infinitely fast at each time hit, and simulation steps do not have a role in a programming model.

We do not take into account the possibility of overflow of timers. Run-time exceptions need to be handled separately.

For validation, we have checked classical properties (deadlock and livelock freedom, and absence of nondeterminism) and analysed timing aspects. The validation has consisted in initially converting *CircusTime* to CSP to use the FDR2 model checker, following a strategy similar to [19], where timing aspects

are captured according to [24].[2] Using the CSP animator, we have observed that at time 0 all inputs are performed in any possible order. After that, internal calculations take place generating the observable output through channel Y. At this point, time (represented by a *tock* event) has to pass before the previous input and output behaviour occurs again. This pattern repeats itself as expected.

As far as we know, there is no report in the literature of support for formal program verification that takes into account the way in which the time model of Simulink diagrams are adapted. Moreover, we are not aware of any formalisation of the typical assumptions embedded in the programming timing model of implementations of control law diagrams.

Timed models of Simulink diagrams are also considered in [14], where a notation called SPI is used to capture control flow. SPI is based on communicating processes; it does not incorporate data operations, but supports the specification of timing restrictions. Since the idealised model of Simulink is not relevant to implementations, the proposed approach is the formalisation of timing requirements after the translation to SPI. It allows the specification of (mode-dependent) data rates and latency times. Using the timed model, it is possible to use static analysis to tackle scheduling. Here, we propose the automatic generation of models.

The approach in [16] uses an extension of Simulink to specify real-time interactions. It is based on a programming language called Giotto. The extended model is translated to Simulink, and then to a program that combines the result of the Simulink code generator with a Giotto program that handles the scheduling. This program runs in an embedded machine that is platform dependent. In our approach, assumptions related to real-time programming are captured using *CircusTime* constructs, and they are uniformly specified (by an *Interface* process) for all applications to be deployed in a particular platform.

The combined use of UML and Simulink is supported by the work in [10], a technique to verify real-time properties of a distributed design compositionally using model checking. It is also part of the trend to verify models and designs, and rely on code generators for the automatic production of programs [11].

The work in [6] also proposes the characterisation of timing requirements based on a calculated model of Simulink diagrams. In that case, the modelling language is TIC (Timed Interval Calculus), a Z-based notation that supports the definition of total functions of continuous time, and (sets of) time intervals. Blocks are specified by schemas, like in ClawZ, but their components include functions from time that define how inputs vary with time, and how outputs are related to inputs over time intervals. Both continuous and discrete times are considered, but not concurrency. The objective of the work is the analysis of diagrams; tool support based on PVS is provided.

A first important piece of future work is the extension of the tool in [31] to automate the generation of our *CircusTime* models and enable significant case studies. For refinement, we will investigate a strategy that transforms the timed

[2] FDR2 is a refinement-checking software tool that can check whether one CSP process refines another, or that a process is free from deadlock, livelock, or nondeterminism. More details about the tool are available from `www.fsel.com/`.

models described here into synchronisation-based models similar to those used in [4]. In that way, we can reuse the verification approach in that work.

A more foundational piece of future work is related to our use of angelic nondeterminism. The semantics of *Circus* and *CircusTime* is defined using Hoare and He's Unifying Theories of Programming (UTP) [13]. In [5], we describe a UTP model for angelic nondeterminism. It remains for us to investigate the consequences of the integration of that model with the *CircusTime* model with deadlines, and to propose and prove laws to support a refinement strategy.

Acknowledgements. Ana Cavalcanti is grateful for the support of EPSRC (grant EP/E025366/1). The work of Alexandre Mota was partially supported by INES11, funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08, by CNPq grant 482462/2009-4 and by the Brazilian Space Agency (UNIESPACO 2009).

References

1. Arthan, R., Caseley, P., O'Halloran, C.M., Smith, A.: ClawZ: Control laws in Z. In: Proceedings of the 3rd IEEE International Conference on Formal Engineering Methods, ICFEM 2000, York, September 4-7, pp. 169–176. IEEE Computer Society, IEEE Press (2000)
2. Boström, P., Morel, L., Waldén, M.: Stepwise development of simulink models using the refinement calculus framework. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 79–93. Springer, Heidelberg (2007)
3. Cavalcanti, A., Clayton, P., O'Halloran, C.: Control law diagrams in *Circus*. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 253–268. Springer, Heidelberg (2005)
4. Cavalcanti, A.L.C., Clayton, P., O'Halloran, C.: From control law diagrams to Ada via *Circus*. Formal Aspects of Computing 23(4), 465–512 (2011)
5. Cavalcanti, A.L.C., Woodcock, J.C.P., Dunne, S.: Angelic nondeterminism in the Unifying Theories of Programming. Formal Aspects of Computing 18(3), 288–307 (2006)
6. Chen, C., Dong, J.S., Sun, J.: A formal framework for modeling and validating Simulink diagrams. Formal Aspects of Computing 21(5), 451–484 (2009)
7. Denney, E., Fischer, B.: Generating customized verifiers for automatically generated code. In: Smaragdakis, Y., Siek, J.G. (eds.) Proceedings of the 7th International Conference on Generative Programming and Component Engineering, GPCE 2008, Nashville, October 19-23, pp. 77–88. ACM (2008)
8. Feliachi, A., Gaudel, M.-C., Wolff, B.: Isabelle/*Circus*: A process specification and verification environment. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 243–260. Springer, Heidelberg (2012)
9. Feliachi, A., Wolff, B., Gaudel, M.-C.: Isabelle/Circus. Archive of Formal Proofs (2012), `http://afp.sourceforge.net/entries/Circus.shtml`
10. Giese, H., Hirsch, M.: Modular verification of safe online-reconfiguration for proactive components in mechatronic UML. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 67–78. Springer, Heidelberg (2006)
11. Graf, S., Gérard, S., Haugen, Ø., Ober, I., Selic, B.: Modelling and analysis of real time and embedded systems – Using UML. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 126–130. Springer, Heidelberg (2007)

12. Hoare, C.A.R.: Communicating Sequential Processes. Series in Computer Science. Prentice Hall International (1986)
13. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Series in Computer Science. Prentice Hall (1998)
14. Jersak, M., Ziegenbein, D., Wolf, F., Richter, K., Ernst, R., Cieslog, F., Teich, J., Strehl, K., Thiele, L.: Embedded system design using the SPI Workbench. In: Proceedings of the 3rd International Forum on Design Languages (2000)
15. Joshi, A., Heimdahl, M.P.E.: Model-based safety analysis of Simulink models using SCADE Design Verifier. In: Winther, R., Gran, B.A., Dahll, G. (eds.) SAFECOMP 2005. LNCS, vol. 3688, pp. 122–135. Springer, Heidelberg (2005)
16. Kirsch, C.M., Sanvido, M.A.A., Henzinger, T.A., Pree, W.: A Giotto-based helicopter control system. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 46–60. Springer, Heidelberg (2002)
17. The MathWorks, Inc., Simulink, `http://www.mathworks.com/products/simulink`
18. Morgan, C.C.: Programming from Specifications, 2nd edn. Prentice Hall (1994)
19. Mota, A.C., Sampaio, A.C.A.: Model-checking CSP-Z: strategy, tool support and industrial application. Science of Computer Programming 40, 59–96 (2001)
20. Oliveira, M.V.M., Cavalcanti, A.L.C., Woodcock, J.C.P.: A UTP semantics for *Circus*. Formal Aspects of Computing 21(1-2), 3–32 (2009)
21. Oliveira, M., Cavalcanti, A., Woodcock, J.: Unifying theories in ProofPower-Z. Formal Aspects of Computing 25(1), 133–158 (2013)
22. Reed, G.M., Roscoe, A.W.: A timed model for communicating sequential processes. Theoretical Computer Science 58, 249–261 (1988)
23. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall Series in Computer Science (1998)
24. Schneider, S.: Concurrent and Real-time Systems: The CSP Approach. Wiley (2000)
25. Sherif, A., He, J.: Towards a time model for *Circus*. In: George, C., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 613–624. Springer, Heidelberg (2002)
26. Sherif, A., He, J., Cavalcanti, A., Sampaio, A.: A framework for specification and validation of real-time systems using *Circus*actions. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 478–493. Springer, Heidelberg (2005)
27. Sherif, A., Cavalcanti, A.L.C., He, J., Sampaio, A.C.A.: A process algebraic framework for specification and validation of real-time systems. Formal Aspects of Computing 22(2), 153–191 (2010)
28. Woodcock, J., Cavalcanti, A.: A concurrent language for refinement. In: Butterfield, A., Strong, G., Pahl, C. (eds.) 5th Irish Workshop on Formal Methods, IWFM 2001, Dublin, July 16-17, BCS, Workshops in Computing (2001)
29. Woodcock, J., Cavalcanti, A.: The semantics of *Circus*. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 184–203. Springer, Heidelberg (2002)
30. Woodcock, J.C.P., Davies, J.: Using Z—Specification, Refinement, and Proof. Prentice-Hall (1996)
31. Zeyda, F., Cavalcanti, A.: Mechanised Translation of Control Law Diagrams into Circus. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 151–166. Springer, Heidelberg (2009)

Concept Analysis Based Approach to Statistical Web Testing

Chao Chen[1,2], Huaikou Miao[1,2], and Yihai Chen[1,2]

[1] School of Computer Engineering and Science, Shanghai University
200072, Shanghai, China
[2] Shanghai Key Laboratory of Computer Software Evaluating and Testing
201112, Shanghai, China
{kingsochen,hkmiao,yhchen}@shu.edu.cn

Abstract. In recent years, the increased complexity of Web applications gives researchers an enormous challenge in ensuring their reliability. A statistical testing approach has been proved to be appropriate to test Web applications and to estimate their reliability. Based on a Markov usage model built from user sessions, testers can generate abstract test cases randomly. In order to execute the test cases, researchers proposed an approach to building a data model from user sessions to provide test data for abstract test cases. However, just building a data model from all the user sessions may cause unauthorized access to restricted pages by testers. In this paper, we propose a concept analysis based approach to statistical Web testing and present a tool to support our approach. We have designed three algorithms to implement our concept analysis based approach that can provide the concrete test data for abstract test cases. In addition, a case study is introduced to demonstrate the effectiveness of our approach to statistical Web testing.

Keywords: statistical testing, Markov usage model, reliability measurement, concept analysis.

1 Introduction

A Web application is a software application that is accessible via a thin client (i.e., Web browser) over a network such as the Internet or an Intranet [1]. Nowadays, Web applications have been widely used to support a variety of activities in our daily lives, such as e-business, online education, online banking, etc. In order to improve the reliability of Web applications, several testing approaches have been proposed by the researchers in the recent years. Coverage based testing and fault based testing are the two strategies of the existing testing approaches. Coverage based testing is to cover a specification to a certain degree [2]. Fault based testing is to design test data to demonstrate the absence of a set of pre-specified faults [2]. Web applications have some characteristics such as massive user population, diverse usage environments, document and information focus [3]. Because of these characteristics, the cost of testing a Web application is too high. Different from the coverage based testing and fault based testing, statistical testing allows testers to focus on the parts of the Web application under test that are frequently

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 100–117, 2013.

accessed [3]. Using statistical testing approaches to test those components that are utilized frequently by the huge number of users is cost-effective. The usual strategy of statistical Web testing has three main steps [3]:

- **Step 1**. Construct a usage model by analyzing the usage information of a Web application.
- **Step 2**. Generate test cases from the usage model and execute them.
- **Step 3**. Collect and analyze the test results to estimate the reliability of the Web application.

To execute the test cases generated in Step 2, testers need to design test data. To reduce the efforts of testers, researchers have proposed some practical approaches. Elbaum et al. [4,5] proposed an approach to obtaining name-value pairs as test input data from user sessions. Elbaum's user session based approach uses a user session as a test case directly or combines multiple user sessions to form a new test case. In their approach, massive user sessions are required to cover the test requirements. Reusing session data as test data is a sensible way, but they did not apply this technique to statistical Web testing. Sant et al. [6] proposed the statistical model for Web testing. In their approach, a control model is built to generate abstract test cases. To provide the test data for abstract test cases, two types of data models are built from user sessions.

The massive user population and increased diversity of Web users result in different usage or multiple roles for a Web application. Distinguishing different roles and their accessible Web pages is a major challenge for the testers if the specification or design documents are not available, but we found testers are still required to log into the Web application to access Web pages. We have attempted to use Sant's approach to providing test data for abstract test cases, but we found testers are always denied access to some restricted pages. The main reason for this issue is that testers use an inappropriate username and password to log into the Web application. In this paper, we propose a concept analysis based approach that combines the data model technique with concept analysis to provide concrete test data for the generated test cases. In particular, a real username and password are included in these test data for testers to log in with an appropriate user role.

The rest of the paper is organized as follows: Section 2 discusses the related work on statistical Web testing. Section 3 presents our concept analysis based approach to generating test data in detail. Section 4 presents the approach to estimating the reliability of a Web application. Section 5 gives a case study of an online bookstore to demonstrate the effectiveness of our approach. Section 6 concludes and proposes further work.

2 Related Work

Many Web testing approaches, such as data flow testing [7], user session based testing [4,5] and URL-based testing [8], have been proposed to support improving the reliability of Web Applications. Statistical Web testing is more cost-effective than traditional approaches to estimate the reliability of a Web application according to some reliability models. In the recent years, many works have been done on statistical Web testing.

Tian et al. [3,9,10] first proposed a statistical testing approach to testing Web applications. In their approach, they analyzed the access log file and constructed UMMs

(Unified Markov Models) as a usage model, which described all the possible usage scenarios of a Web application. Test cases were generated randomly from UMMs, selected and executed according to the threshold probability. The results of testing were stored in the server error log file. By analyzing the access log file and the error log file, they utilized Nelson model [11] and G-O model [12] to estimate the overall reliability of a Web application.

Hao et al. [13] undertook two experiments to confirm the effectiveness of UMMs. Different from Tian's experiment, they used multiple set of UMMs in their experiments because the entries of a Web application were always more than one. As the experiments' results shown, UMMs are suitable and adequate to be used as usage models for statistical Web application.

Tonella and Ricca [14,15] proposed a dynamic analysis approach to extract Web application model for an existing Web application. The node of their model represents a static Web page, server program or dynamic page, and the edge represents the interaction behaviors between two nodes. In their approach, the Web pages were regarded as states, and the hyperlinks of states were regarded as transitions. They built Markov chain as the usage model in order to support statistical Web testing. The nodes and edges were same as the Web application model extracted by dynamic analysis. The probabilities of transitions could be obtained by analyzing the access log file. They executed the test cases generated from usage model to estimate the reliability of a Web application. Although the purpose of their work was same as Tian's work and Hao's work, the definitions of Web failures were different from Tian's work and Hao's work. In both Tian's work and Hao's work, they estimated the operation reliability of a Web application, and the failures were stored in the error log file. Tonella and Ricca defined the Web failures as the output pages that are different from the expected pages, and these failures could not be stored in the error log files.

Sant et al. [6] proposed an approach to building statistical models of user sessions and generating test cases from these models automatically. In order to model the Web applications, they designed a control model and a data model. The control model was represented by the possible sequences of URLs, and the data model was represented by the possible sets of parameter values.

Sprenkle et al. [16] did a study of constructing the navigation model directly from user sessions. In their work, the abstract test cases, represented by the sequences of states, are generated automatically. The states of navigation models can be represented by RR (Request Type + Resource), RRN (Request Type + Resource + Parameter Names) and RRNV (Request Type + Resource + Parameter Names + Values). The different representations have an effect on the numbers of states and transitions of the navigation model.

For the given abstract test cases, our concept analysis based approach can provide concrete test data including the real username and password for testers to log in with an appropriate user role. In our approach, a concept lattice is constructed from all the user sessions and a data model is built for each concept. The test data can be generated from its data model according to the condition probability. Moreover, we have developed a tool to support our approach.

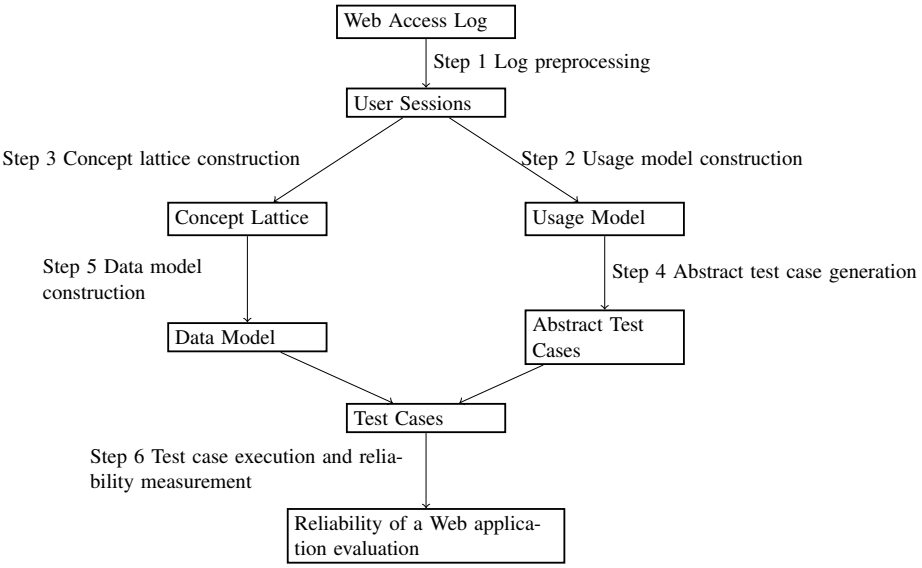


Fig. 1. Complete Process of Our Approach

3 Concept Analysis Based Statistical Web Testing

Using data model represented by the possible sets of name-value pairs is an effective approach to providing test data for the abstract test cases. The access log file consists of huge a number of user sessions, and reflects the navigation patterns of different roles. As a result, the data model, built from all the user sessions, may provide inappropriate username and password for testers to log into the Web application. By analyzing one user's sessions, we can know the pages which have been accessed successfully by the user. Therefore, Web users are allowed to access the pages which are included in their user sessions. We assume that u_i is the user of the i_{th} user session US_i, and R_{ui} denotes the set of the Web pages which are allowed to be accessed by u_i. R_{usi} is used to denote the set which consists of the Web pages of session us_i, and R_t is used to denote the set which consists of the Web pages of the given abstract test case tc. The property we found can be described as follow.

$$\forall us_i \in US_i, \forall r_t \in R_t \ (R_{usi} \supseteq R_t \Rightarrow r_t \in R_{ui})$$

Based on the above property, we propose a concept analysis based statistical Web testing approach. In our approach, concept analysis is used to cluster the user sessions into a concept which contains all the Web pages of one generated test case. The complete process of our approach is shown in Fig. 1. Step 1 is to identify the user sessions from the access log file. In Step 2, these user sessions are used to construct the Markov usage model which describes all the users' navigation patterns. The user sessions can also be used to build a concept lattice in Step 3. The concept lattice depicts the relationship between user session and Web pages. The abstract test cases are generated from Markov

usage model in Step 4, but these test cases are represented by state sequences and cannot be executed directly. In Step 5, we build one data model for each concept, and the data model is built from the user sessions of the corresponding concept. In each data model, the name-value pairs have the condition probabilities, and they can be used as the test data for the abstract test cases. In Step 6, these test cases are executed by the replay tool Httpclient [17]. After executing test cases, we analyze the test result using the reliability models to estimate the reliability of a Web application.

3.1 Access Log Preprocessing

Web access log is a file which has stored all the users' requests via GET or POST method. An example of a log entry is shown as follows.

```
192.168.0.102 - - [06/Jan/2013:15:03:23 +0800] "GET /books
tore2/MyInfo.jsp HTTP/1.1" 200 7037 "http://192.168.0.101:
8080/bookstore2/ShoppingCart.jsp" "Mozilla/4.0 (compatible;
MSIE 8.0; Windows NT 6.1)"
```

In above example, "192.168.0.102" is the IP address of requesting computer. "06/Jan/2013:15:03:23 +0800" is the Date and Time of the request. "MyInfo.jsp" is the Resource which Web users request. "ShoppingCart.jsp" is the Referrer URL. "Mozilla/4.0" is the information of User-Agent.

We define the access log entry as an 8-tuple $\langle ip,\ time,\ preSt,\ nextSt,\ agent,\ formName,\ formAction,\ userName\rangle$. In this tuple, ip denotes user's ip address; $time$ denotes the date and time when user request the Web application; $preSt$ denotes the state before Web users request the Web application; $nextSt$ denotes the next state. $agent$ denote the browser information of users. $formName$ and $formAction$ denote the name and the action of a form. $userName$ denotes the username of a Web user. If a log entry does not contain the value of username, the $userName$ will be set to "NULL". The user session can be regarded as several log entries for one user in the period of time. We assume that LS denotes the set which include all the log entries and U denotes the set which includes all the Web users, then each user session can be defined as:

US = $\langle u, ls\rangle$
u denotes the Web user, $u \in U$.
$u = \langle ip, agent, userName\rangle$. ip is the IP address of a Web user,
$agent$ denotes the User Agent.
$userName$ is the username when a user signs in the Web application.
ls is the set of log entries, $ls \subseteq LS$.

The goal of log preprocessing is to identify the Web users and the user sessions. The method of user identification and session identification is same as Sant's work [6]. The implementation of their method can be described as algorithm 1. In algorithm 1, Step 1 is used to identify the user from huge numbers of Web users. By user identification, we have known all the users and the corresponding set of log entries, denoted as L_u. One user's log entries within a period of time can be regarded as a user session. Step 2

is used to identify the user sessions for one user. After session identification, we obtain the user sessions of the user u. We group these sessions of user u into a set US_u. Step 3 is used to assign the username to a user of each user session.

Algorithm 1. Preprocessing the access log file.

Step 1. Finding out the log entries for a given user u and grouping them into a set of log entries L_u.

For each log entry l_i in the log file
  If ($l_i.ip == u.ip \wedge l_i.agent == u.agent$)
    $L_u = L_u \cup \{l_i\}$
  End if
End for

Step 2. Session identification
For two consecutive log entries l_i and l_{i+1} in L_u
  If ($l_{i+1}.time - l_i.time < interval\ time$)
    $ls = ls \cup \{l_i, l_{i+1}\}$
  Else
    $ls = ls \cup \{l_i\}$
    $us = \langle u, ls \rangle$
  End if
End for

Step 3. Assigning the username to each session user.
For each us_i in US_u,
(we assume ls_i denotes the set of log entries of us_i, and u_i is the user of us_i)
  If ($\exists l_i \in ls_i\ (l_i.userName \neq NULL)$)
    $u_i.username = l_i.userName$
  End if
End for

3.2 Building Usage Model

Both flat operation profile [18] and Markov usage model [19] can be used to represent the complete usage patterns of Web users. Markov usage model is a flexible, easily understandable descriptions of the operational profiles, and it provides both a wide range of applications and a potential for deep formal analysis [19]. Building Markov usage model is an important step of the statistical Web testing because the abstract test cases are generated from the Markov usage model. In our approach, each Web page is regarded as a state of Markov usage model. Two types of transitions, hyperlink transitions and form data submission transitions, are included in our Markov chain. The form data submission transitions can be distinguished from the hyperlink transitions by $formName$ and $formAction$. A Markov usage model can be defined as [20]:

$MC = \langle S,\ \Gamma,\ \delta,\ s_0, f \rangle$
S is the set of states,
Γ is the set of transition labels,
δ: $S \times \Gamma \rightarrow S$ is next state function,

s_0 is the initial state,
f is the final state.

The transition label $t \in \Gamma$ is defined as a tuple $\langle preSt,\ nextSt,\ count,\ prob,\ formName,\ formAction\rangle$. $preSt$ is the present states of a transition, and $nextSt$ is the next state of a transition. The probability of the label t is denoted by $prob$, the variable $count$ denotes the execution counts. The variables $formName$ and $formAction$ denote the name and the action of a form.

A Markov chain can be represented by a direct graph which contains several nodes and edges. The nodes represent the states (i.e., Web pages) of the Web application, and the edges represent the transitions of states (i.e., links). Each edge has a transition probability ranging from zero to one. The construction method of a Markov usage model is same as Sant's control model [6] and Sprenkle's navigation model [16]. We design an algorithm to implement the method. This algorithm builds the Markov usage model from a Web access log file.

Algorithm 2. Building the Markov usage model.

Step 1. Scan all the user sessions and get the set of states S.
For each log entry l_i,
  If ($\nexists s \in S(s == l_i.preSt)$)
    $S = S \cup \{preSt\}$
  End if
  If ($\nexists s \in S(s == l_i.nextSt)$)
    $S = S \cup \{nextSt\}$
  End if
End for
Step 2. Get the set of transition labels.
For each log entry l_i,
  If ($\nexists t \in \Gamma$ ($t.preSt == l_i.preSt \wedge t.nextSt = l_i.nextSt \wedge$
  $t.formName == l_i.formName \wedge t.formAction == l_i.formAction$))
    create a new transition label $t' \wedge t'.preSt = l_i.preSt \wedge$
    $t'.nextSt = l_i.nextSt \ \wedge t'.count = 1 \wedge$
    $t'.formName = l_i.formName \wedge$
    $t'.formAction = l_i.formAction \wedge$
    $\Gamma = \Gamma \cup \{t'\}$
  Else
    $t.count = t.count$ + 1;
  End if
End for
Step 3. Calculate the probability of each transition label t_i.
For each transition label t_j,
  If ($t_i.preSt == t_j.preSt$)
    $count_{total} = count_{total} + t_j.count$
  End if
End for
$t_i.prob = t_i.count/count_{total}$

3.3 Concept Lattice Construction

Concept analysis is a mathematical technique for clustering objects that have common discrete attributes [21]. The input of concept analysis consists of a set of objects, a set of attributes and the relationship between the objects and the attributes. A concept is defined as a tuple $C = \langle O_i, A_j \rangle$, and it means all and only objects in O_i share all and only attributes in A_j [21]. The relationship between two concepts has been called partial order. The partial order of two concepts can be defined as: $(O_1, A_1) \subseteq (O_2, A_2) \iff O_1 \subseteq O_2$. Being different from Sampath's work [21], we apply concept analysis to support statistical Web testing rather than reduce the size of a test suite. In our statistical Web testing approach, we use concept analysis to find the user sessions which contain all the states of abstract test cases, and extract the name-value pairs as the test data for the abstract test cases. The user sessions can be regarded as the objects, and the states (i.e., Web pages) of the abstract test cases can be regarded as the attributes of the objects.

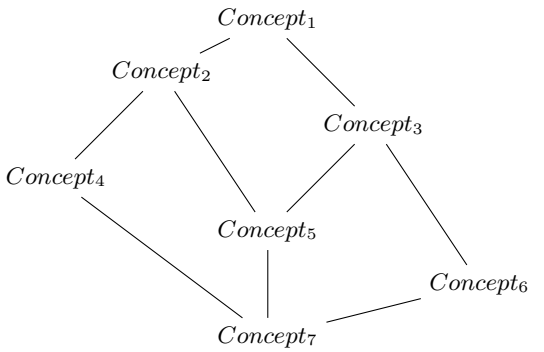
The concepts and their partial ordering constitute a concept lattice. We adapt the Chein's algorithm [19] to construct a concept lattice which has multiple layers, and each layer has several different concepts. The process of Chein's algorithm is creating new concepts for a higher layer by the intersection of two concepts in the same lower layer, and removing the redundant concepts in the lower layer. A concept lattice can be represented by a Hasse diagram. In a Hasse diagram [21], the concept in the top node is most general. All the attributes of this concept are shared by all the objects in the whole concept lattice [21]. Similarly, the concept in the bottom node is most special, and all the objects of this concept have all the attributes in the whole concept lattice [21]. A sample of Hasse diagram, which represents the relationship between objects and attributes in Table 1, is shown in Fig. 2. $Concept_1$ is a concept in the top node, and all the sessions have all the states (i.e., $us_1, us_2, ..., us_7$) in the attribute set (i.e., $\{Login\}$) of $Concept_1$. $Concept_7$ is a concept in the bottom node, and the object (i.e., us_7) in its object set has all the attributes (i.e., $Login$, $ShoppingCart$, $ShoppingCartRecord$, $BookDetail$, $Default$).

Table 1. A Sample of User Sessions and Their States

Sessions/States	Login	ShoppingCart	Default	ShoppingCartRecord	BookDetail
User Session 1	1	0	1	1	1
User Session 2	1	1	0	1	0
User Session 3	1	0	1	0	0
User Session 4	1	1	0	1	0
User Session 5	1	0	1	0	1
User Session 6	1	1	1	0	0
User Session 7	1	1	1	1	1

3.4 Abstract Test Cases Generation

The test cases generation phase of statistical Web testing is different from traditional Web coverage-based testing [22]. In statistical Web testing, abstract test cases [16] can



$Concept_1$: $(\{us_1, us_2, us_3, us_4, us_5, us_6, us_7\}, \{Login\})$
$Concept_2$: $(\{us_2, us_4, us_6, us_7\}, \{Login, ShoppingCart\})$
$Concept_3$: $(\{us_1, us_3, us_5, us_6, us_7\}, \{Login, Default\})$
$Concept_4$: $(\{us_2, us_4, us_7\}, \{Login, ShoppingCart, ShoppingCartRecord\})$
$Concept_5$: $(\{us_1, us_5, us_7\}, \{Login, BookDetail, Default\})$
$Concept_6$: $(\{us_6, us_7\}, \{Login, ShoppingCart, Default\})$
$Concept_7$: $(\{us_7\}, \{Login, ShoppingCart, ShoppingCartRecord, BookDetail, Default\})$

Fig. 2. A Sample of Hasse Diagram

be generated from the Markov usage model by a random traverse. The generated abstract test cases are represented by state sequences, which cannot be executed directly. Each abstract test case has its probability, and the abstract test case probability ptc_i can be calculated by the following equation which is proposed by Walt J [19]:

$$ptc_i = \prod_{j=1}^{k} pt_j \tag{1}$$

In Equation (1), the abstract test case tc_i has k transitions and the variable pt_j is the probability of the jth transition. For example, the probability of the abstract test case "Start $\rightarrow$ Login $\rightarrow$ Registration $\rightarrow$ Default $\rightarrow$ Exit", according to the Markov usage model shown in Fig. 3, can be calculated as $1 \times 0.476 \times 0.692 \times 0.087 = 0.0286$.

The threshold probability is set for the process of abstract test cases selection. If the probability of one abstract test case is larger than the threshold probability, this abstract test case will be added into the test suite. In the process of generating the abstract test cases, the threshold probability is adjusted dynamically to control the numbers of abstract test cases.

3.5 Data Model Construction and Obtaining Test Data

The purpose of data model construction is to provide the test data for the abstract test cases when testers execute these test cases. A data model is built for each concept in the concept lattice constructed previously. The user sessions of a concept are extracted, and a data model can be built based on these user sessions. In our approach, the data

model, shown as follow, can be represented by the conditional probability of the given $userName$, $lastPage$, $visitingPage$, $formName$ and $formAction$.

$$P(name\_values \mid userName,\ lastPage,\ visitingPage + formName + formAction)$$

When providing the test data for one abstract test case, we need to search the concept whose attribute set (i.e., the set of states) contains all the states of the given test case. We obtain the name-value pairs as the test data according to their condition probabilities.

Different from Sant's work, we build one data model for each concept from this concept's user sessions. Our approach makes testers log into the Web application with an appropriate role to execute one test case. Because the test data, including the username and password, is obtained by the sessions whose users are allowed to access all the pages of this test case. In addition, the username is taken into account in our data model. This ensures that the test data is obtained from one user's session.

We design algorithm 3 to describe finding the suitable concept for the given abstract test case from the concept lattice. In algorithm 3, the set of the states of one abstract test case is denoted as S_t. We assume the concept lattice has n layers, and the nth layer is the top layer.

Algorithm 3. Searching the suitable concept for the given abstract test case.

Step 1. Current layer $k = n$.

Step 2. For each C_i in layer k
  The attribute set of C_i is denoted as S_i
  If ($S_t \subseteq S_i$)
    C_i is the suitable concept.
    Stop algorithm.
  End for

Step 3. If no suitable concept has been found in layer k
  $k = k - 1$.
  Goto Step 2.
End if

4 Reliability Measurement

The reliability of Web applications can be defined as the probability of failure-free Web operation completions [23]. Nelson model [11] is one of the classic reliability models which can be used to measure the reliability of Web applications. To apply Nelson model, usage information and failure information are required. The usage data and error data are stored by Web server automatically. When a Web page or any other Web resource, such as a sound file, a video file, an image file, etc, is requested by Web user, a "hit" will be registered in the access log file. When a failure occur, the failure information is stored in the error log file. After executing the test cases, we can analyze the error log file to obtain the failure information. Some errors stored in the error log file are shown as follows.

```
[ERROR] http-8080-3 org.apache.catalina.core.ContainerBase.
[Catalina].[localhost].[/bookstore2].[jsp] -
{Servlet.service () for servlet jsp threw exception}
java.sql.SQLException: Access denied for user 'root1'@
'localhost'  (using password: YES)

[ERROR] http-8080-1 org.apache.catalina.core.ContainerBase.
[Catalina].[localhost].[/bookstore2].[jsp] -
{Servlet.service () for servlet jsp threw exception}
com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException:
Unknown database 'bookstore'
```

We can detect the existent errors of a Web application by analyzing the error information. The first error results from using incorrect username or password to connect the database. The crucial reason for second error is to connect the unknown database. These errors can be regarded as the failures of a Web application. In order to estimate the overall reliability of a Web application, we need to count the total number of errors, denoted as f. According to Nelson model [11], for total hits n, the estimated reliability R can be calculated as the following equation:

$$R = 1 - \frac{f}{n} \tag{2}$$

According to Tian's work [3], MTBF (Mean Time between Failures) can be calculated as:

$$MTBF = \frac{n}{f} \tag{3}$$

5 Case Study

In this section, we present and discuss statistical testing of a bookstore Web application, which is downloaded from gotocode.com as an empirical study. Our experiment is deployed in a LAN, and we use Tomcat 6.0 as our Web Server. We invite our classmates and Ph.D students as volunteers to be the Web users. All the users' requests will be stored automatically in the access log.

5.1 Access Log Preprocessing

The access log file we analyzed contains 2,035 log entries. We remove the image, video and sound request log entries from this access log file. We identify the Web users by the step 1 of algorithm 1. The log entries belongs to a user with the same IP address and the same User Agent. We implement the session identification by step 2 of algorithm 1 to discover the user sessions. The interval time in session identification is fixed to 10 minutes. As a result, we obtain 35 user sessions from session identification.

5.2 Usage Model Construction

As mentioned in Section 3.2, Markov usage model is more appropriate than flat operation profile to represent the whole navigation patterns. Therefore, we use Markov chain as the usage model for the Web application. The Markov usage model can be constructed by algorithm 2. In this case study, the Markov usage model has 25 states and 78 transitions. Fig. 3 shows part of the Markov usage model of our Web application under test (i.e, some transitions and states are not shown in Fig. 3, and the summation of transition probabilities from one state may not equal to one).

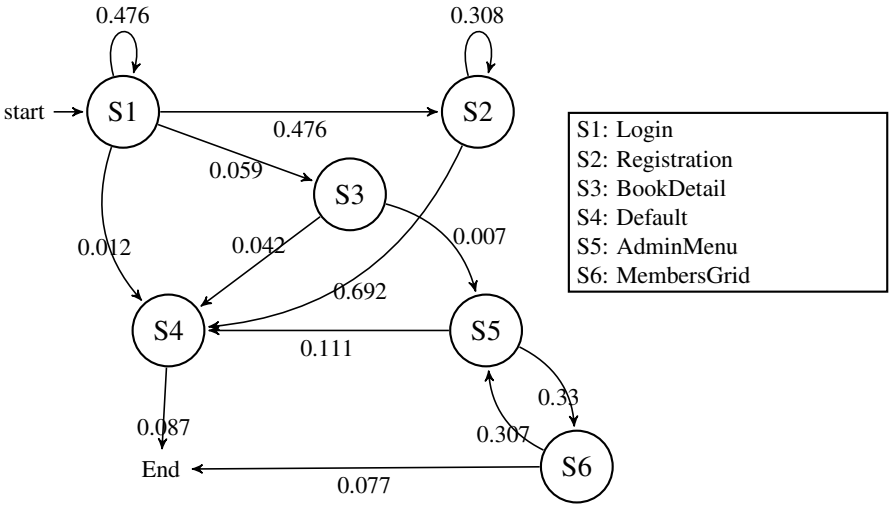


Fig. 3. Part of the Markov chain usage model in our experiment

5.3 Abstract Test Cases Generation

The abstract test cases can be generated from the Markov chain by random traverse (i.e., choosing the next state randomly). We set 0.1 as the initial threshold probability to control the numbers of abstract test cases. This threshold probability will be adjusted dynamically during the test case generation process. Finally, we generate 500 unique abstract test cases. As shown in the following, the abstract test case No. 448 is an example in our experiment.

Abstract Test Case No. 448:
Start
→ /bookstore2/Login.jsp → /bookstore2/Login.jsp_Form_Login_login
→ /bookstore2/BookDetail.jsp → /bookstore2/AdminMenu.jsp
→ /bookstore2/MembersGrid.jsp → /bookstore2/MembersRecord.jsp
→ /bookstore2/MembersRecord.jsp_Form_Members_insert
→ /bookstore2/MembersGrid.jsp → /bookstore2/MembersRecord.jsp
→ /bookstore2/MembersRecord.jsp_Form_Members_insert
→ /bookstore2/MembersGrid.jsp → /bookstore2/Default.jsp
→ End

As shown above, abstract test case No. 448 contains some restricted operations such as visiting an administrator's menu, adding members, etc. In "`MembersRecord.jsp_Form_Members_insert`", "`Form`" means the transition from "`MembersRecord.jsp`" to "`MembersGrid.jsp`" need to submit the form data. "`Members`" is the name of the form, and "`insert`" is the action of this form.

5.4 Providing Test Data for Abstract Test Case by Concept Analysis

We use concept analysis to cluster the user sessions which has the same Web pages (i.e., states of Web application) into a group. In concept analysis, each concept has a set of objects and these objects share common attributes. In our experiment, the user sessions are treated as objects, and the Web pages are the attributes of these objects. A concept lattice describes the relationship of concepts, and it can be represented by a Hasse diagram. In our experiment, 35 user sessions found in session identification can be treated as 35 objects, and these sessions contain 25 states. These states are treated as 25 attributes. We construct the concept lattice using Chein's algorithm [24]. The concept lattice in our experiment contains 41 concepts and 8 layers.

In order to obtain the session data (i.e., name-value pairs) for an abstract test case, we search the concept, which contains all the states of this abstract test case, from the concept lattice. We use the name-value pairs as the test data according to their condition probability in the data model of this concept. The test case No. 448, including test data, is shown as follows.

Test Case No. 448:
Start
→ /bookstore2/Login.jsp
→ /bookstore2/Login.jsp?FormName=Login&Login=admin&Password=admin&
FormAction=login&ret_page=&querystring=
→ /bookstore2/BookDetail.jsp → /bookstore2/AdminMenu.jsp
→ /bookstore2/MembersGrid.jsp → /bookstore2/MembersRecord.jsp?name=&
→ /bookstore2/MembersRecord.jsp?member_login=robert_123
&member_password=robert_pswd ...
→ /bookstore2/MembersGrid.jsp?member_login=
→ /bookstore2/MembersRecord.jsp?name=&
→ /bookstore2/MembersRecord.jsp?member_login=creator
&member_password=creatorpswd ...
→ /bookstore2/MembersGrid.jsp?member_login= → /bookstore2/Default.jsp
→ End

We have developed a Web statistical test prototype tool to support our approach. It can identify user sessions, construct concept lattice, build Markov usage model, generate abstract test cases and test cases, and estimate the reliability of a Web application automatically. As shown in Fig. 4, the test case No. 448 is one of the test cases generated by our testing tool. In this test case, "`AdminMenu.jsp`", "`MembersGrid.jsp`" and "`MembersRecord.jsp`" are accessible only by the administrators (i.e., testers must log into the Web application with an administrator rather than a customer). In order to testify the effectiveness of our approach, we have attempted to use Sant's approach

Table 2. The username and password produced by Sant's data model and by our approach

Abstract Test Case No. 448	Sant's approach			
Login.jsp	username	password	isAdmin	percentage
Login.jsp_Form_Login_login	solar3	sunna88	false	18%
BookDetail.jsp	admin2	admin2	true	15%
AdminMenu.jsp	guest	guest	false	14%
MembersGrid.jsp	admin1	admin1	true	12%
MembersRecord.jsp	admin	admin	true	11%
MembersRecord_Form_Members_insert	lunna	vison	false	10%
MembersGrid.jsp	user2	123456	false	2%
MembersRecord.jsp	...	...	...	...
MembersRecord_Form_Members_insert				
MembersGrid.jsp				
Abstract Test Case No. 448	Our approach			
Login.jsp	username	password	isAdmin	percentage
Login.jsp_Form_Login_login	admin1	admin1	true	36%
BookDetail.jsp	admin	admin	true	31%
AdminMenu.jsp	admin2	admin2	true	33%
MembersGrid.jsp				
MembersRecord.jsp				
MembersRecord_Form_Members_insert				
MembersGrid.jsp				
MembersRecord.jsp				
MembersRecord_Form_Members_insert				
MembersGrid.jsp				

and our approach for 100 times to provide the test data for the given abstract test case No. 448. We can conclude form Table 2 that "admin", "admin1" and "admin2" are the administrators, testers must use their username and password to log into the Web application. If we obtain the test data from the data model built from all the user sessions, the correctness rate is only about 38%. Compared with the results of this approach, the performance of our approach is remarkable better.

The executable test cases generated in our experiment can be executed directly by replay tool, such as Httpclient.

5.5 Reliability Measurement

After the test cases being executed, the failures information is stored in the error log file. We analyze the error log file and count the failures f. For total hits n, the relationship between total hits n and failures f has been shown as Fig. 5. When every 50 test cases being executed, the test results will be analyzed. The reliability of Web application can be estimated according to Equation (2) and (3). Fig. 6 and Fig. 7 show the reliability R and MTBF of this Web application.

Fig. 5 shows the relationship between the cumulative errors and cumulative hits. The slope of curve depicts the frequency of the failures occur. We can conclude from Fig. 6

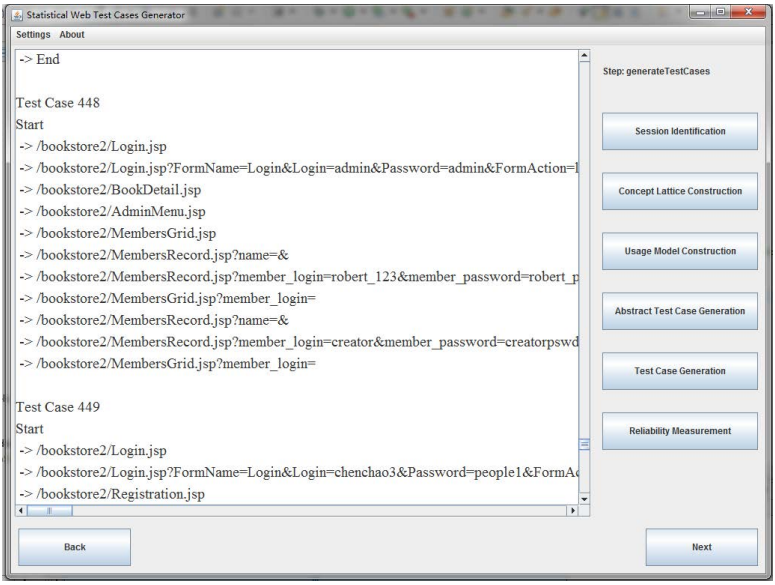


Fig. 4. Statistical Web test cases generator

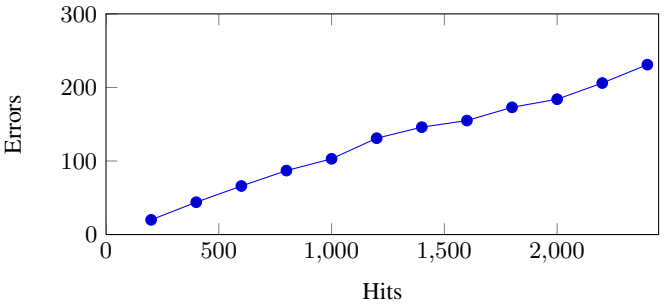


Fig. 5. Hits versus errors

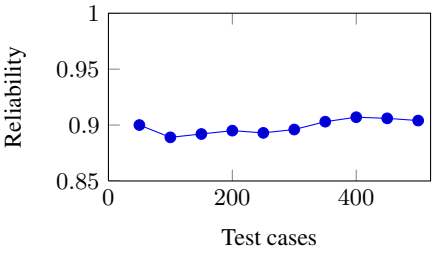


Fig. 6. Relationship between test cases and reliability

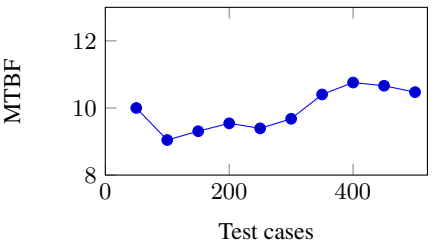


Fig. 7. Relationship between test cases and MTBF

that the reliability is from 88.9% to 90.7%, and Fig. 7 shows that the MTBF of the Web application is from 9.045 hits to 10.759 hits. This means one failure will occur in about 10 hits.

6 Conclusion and Future Work

Statistical testing is a cost-effective testing and appropriate approach to testing Web applications. By statistical testing, the overall reliability of Web applications can be estimated by reliability models. However, most researchers focus on how to build the Markov usage model and generate the abstract test cases from it. It is also important for us to provide the concrete test data, especially the real username and password for testers to log in with an appropriate role, for the abstract test cases.

Through further study, we found that the approach proposed by Sant [6], building the data model to provide test data for the abstract test cases, cannot ensure that the testers have the authority to access to the pages. For example, testers may use a customer's username and password to log in before they try to access to the administrator's menu page. The main reason for this problem is that the data model built from all the user sessions, with users of these user sessions having different levels authority. In our approach, the test data are obtained from the user sessions which include all the pages of a test case. As a result, our approach provides an appropriate username and password for testers to log in, and it ensures that they have the authority to access the pages of a given test case.

The main contribution of this paper has been to propose an approach combining concept analysis with a data model to provide the concrete test data in statistical Web testing. The test data generation is different from existing approaches; we utilize concept analysis to cluster the user sessions which have the same Web pages and construct a data model for each concept. When proving test data, for an abstract test case, we analyze the name-value pairs from the user sessions in one concept and extract some of them as test data according to their condition probabilities. The reliability can be estimated using the Nelson model and MTBF after the test cases are executed by the replay tool. Moreover, a tool has been developed to support our approach to statistical Web testing. The tool allows testing of the Web application and gives an estimate of its reliability.

In our approach, we utilize Chein's algorithm [24] to construct the concept lattice, which describes the relationship of different concepts. However, Chein's algorithm is one of batch algorithms, which costs much time and space. To test a large scale Web application statistically, testers are required to analyze a huge number of user sessions. Therefore, it is necessary for us to use a more efficient algorithm, such as Godin's incremental algorithm [25], to construct the concept lattice. In addition, the Web users' specific operations, such as "back", "refresh", "forward" and entering URL to browse Web application directly, will affect the accuracy of our usage model. In our future work, we will utilize related data mining techniques to improve the accuracy of the Markov usage model. Moreover, only server exceptions and errors can be stored in the error log file. Being similar to Tian's work [3], we estimate the operation reliability of the Web application according to the errors (stored in the error log) and hits. However,

the error log file cannot store some errors of Web applications (e.g., the output page is different from the expected page). In fact, confirming whether the output pages are same as the expected pages is also important and meaningful. In our future work, we will take these errors into account and estimate the reliability of Web applications.

Acknowledgments. This paper is supported by National Natural Science Foundation of China (NSFC) under grant No. 61073050 and 61170044, and by Shanghai Leading Academic Discipline Project (Project Number: J50103).

References

1. Alalfi, M.H., Cordy, J.R., Dean, T.R.: Modelling Methods for Web application Verification and Testing: State of the Art. Software Testing, Verification and Reliability 19, 265–296 (2009)
2. Aichernig, B., He, J.: Mutation Testing in UTP. Formal Aspects of Computing 21, 33–64 (2009)
3. Kallepalli, C., Tian, J.: Measuring and Modeling Usage and Reliability for Statistical Web Testing. IEEE Transactions on Software Engineering 27(11), 1023–1036 (2001)
4. Elbaum, S., Rothermel, G.: Leveraging User-session Data to Support Web Application Testing. IEEE Transactions on Software Engineering 31(3), 187–202 (2005)
5. Elbaum, S., Karre, S., Rothermel, G.: Improving Web Application Testing with User Session Data. In: Proceedings of the 25th International Conference on Software Engineering, pp. 49–59 (2003)
6. Sant, J., Souter, A., Greenwald, L.: An Exploration of Statistical Models for Automated Test Case Generation. In: Proceedings of the Third International Workshop on Dynamic Analysis, pp. 1–7 (2005)
7. Liu, C., Kung, D., Hsia, P.: Object-based Data Flow Testing of Web Applications. In: Proceedings of the First Asia-Pacific Conference on Quality Software, pp. 7–16 (2000)
8. Wen, R.: URL-driven Automated Testing. In: Proceedings Second Asia-Pacific Conference on Quality Software, pp. 268–272 (2001)
9. Tian, J., Koru, A.: A Hierarchical Strategy for Testing Web-based Applications and Ensuring Their Reliability. In: Proceedings of the 27th Annual International Computer Software and Applications Conference, pp. 702–707 (2003)
10. Tian, J.: Testing the suitability of Markov chains as Web usage models. In: Proceedings 27th Annual International Computer Software and Applications Conference, pp. 356–361 (2003)
11. Nelson, E.: Estimating Software Reliability from Test Data. Microelectronics Reliability 17(1), 67–73 (1978)
12. Goel, A.: Software Reliability Models: Assumptions, Limitations, and Applicability. IEEE Transactions on Software Engineering SE-11(12), 1411–1423 (1985)
13. Hao, J., Mendes, E.: Usage-based Statistical Testing of Web Applications. In: Proceedings of the 6th International Conference on Web Engineering, pp. 17–24 (2006)
14. Tonella, P., Ricca, F.: Statistical Testing of Web Applications. Journal of Software Maintenance and Evolution: Research and Practice 16(12), 103–127 (2004)
15. Tonella, P., Ricca, F.: Dynamic Model Extraction and Statistical Analysis of Web Applications. In: Proceedings of the Fourth International Workshop on Web Site Evolution, pp. 43–52 (2002)
16. Sprenkle, S., Pollock, L., Simko, L.: A Study of Usage-Based Navigation Models and Generated Abstract Test Cases for Web Applications. In: Proceedings of the Fourth IEEE International Conference on Software Testing, pp. 230–239 (2011)

17. HttpClient:
http://hc.apache.org/httpcomponents-client-ga/index.html
18. Musa, J.: Operational Profiles in Software-reliability Engineering. IEEE Software 10, 14–32 (1993)
19. Gutjahr, W.: Software Dependability Evaluation Based on Markov Usage Models. Performance Evaluation 40, 199–222 (2000)
20. Yan, J., Wang, J., Chen, H.: Deriving Software Markov Chain Usage Model from UML Model. Journal of Software 16, 1386–1394 (2005)
21. Sampath, S., Mihaylov, V.: A Scalable Approach to User-session Based Testing of Web Applications through Concept Analysis. In: Proceedings of the 19th International Conference on Automated Software Engineering, pp. 132–141 (2004)
22. Zeng, H., Miao, H.: Auto-generating Test Sequences for Web Applications. In: Baresi, L., Fraternali, P., Houben, G.-J. (eds.) ICWE 2007. LNCS, vol. 4607, pp. 301–305. Springer, Heidelberg (2007)
23. Li, Z., Tian, J.: Analyzing Web Logs to Identify Common Errors and Improve Web Reliability. In: Proceedings of the IADIS International Conference E-Society, pp. 235–242 (2003)
24. Tongkun, J.: The Research and Improvement of Concept Lattice Chein Algorithm. Dissertation, South China University of Technology, Guangzhou, China (2012)
25. Godin, R., Missaoui, R., Alaoui, H.: Incremental Concept Formation Algorithms Based on Galois (Concept) Lattices. Computational Intelligence 11(2), 246–267 (1995)

Algebraic Program Semantics for Supercomputing

Yifeng Chen

HCST Key Lab at School of EECS
Peking University
Beijing 100871, China

Abstract. The competition for higher performance/price ratio is pushing processor chip design into the manycore age. Existing multicore technologies such as caching no longer scale up to dozens of processor cores. Even moderate level of performance optimisation requires direct handling of data locality and distribution. Such architectural complexity inevitably introduces challenges to programming. A better approach for abstraction than completely relying on compiler optimization is to expose some performance-critical features of the system and expect the programmer to handle them explicitly. This paper studies an algebra-semantic programming theory for a performance-transparent level of parallel programming of array-based data layout, distribution, transfer and their affinity with threads. The programming model contributes to the simplification of system-level complexities and the answering of crucial engineering questions through rigorous reasoning.

1 Introduction

Driven by the demand for higher performance and lower hardware and energy costs, hardware systems are increasingly heterogeneous and massively parallel. A typical large hybrid system may include thousands of computing nodes, each with a small number of complex CPU cores and a large number of simpler GPU cores for acceleration. Such hybrid solution is power-efficient if massively parallel tasks are properly delegated to the accelerators.

The architectural complexity inevitably introduces challenges to programming. As achieving uniform sequential consistency for hundreds of cores is often too expensive, practical design solutions either expose the on-chip memory hierarchy *e.g.* in NVidia GT200 and 48-core Intel Single-Chip Cloud Computer (SCCC) or implement a non-uniform coherent cache along a ring bus *e.g.* in Larrabee and Knights Corner. A non-uniform cache is transparent for consistency but opaque for performance, which heavily depends on the distances between cores on the ring bus. That means, to achieve reasonable cache performance, the locality of data-thread mapping as well as many other architectural factors must be taken into account by either the user program or the compiler. There has been intensive academic research on unifying memory models that are more relaxed than sequential

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 118–135, 2013.

consistency but provide better performance, although the hardware industry has currently not been pursuing these directions.

A common misperception is that the user program need not handle any system-level features and should leave them to the compiler for optimization — by conceding a fraction of performance, the productivity is improved. New architectural technologies, however, can be much more sensitive to suboptimal implementation. The performance penalty for slight mismanagement of threads or memory can be orders of magnitude. For general-purpose applications, compile-time and runtime optimization alleviates the issues but it remains to be seen to what extent the lost source information about system-level control is recoverable automatically. Experiences also show that engineers often prefer those parallel-programming tools with predictable performance over automated optimizers with unreliable effectiveness.

A more utile approach for abstraction is to expose some performance-critical features of the system and expect the programmer to handle them explicitly. Then the question is how to identify and organize such features in a unified representation that is easy for the programmer to comprehend and control. Designing new language features is a central concern of the programming-language community, although the initial motivation here is performance rather than correctness (which, of course, naturally arises too).

The mission of this paper is to "convert" these emerging engineering challenges to the realm of programming-language theory. We will motivate and show how formalization can contribute to the simplification of system-level complexities and how crucial engineering questions are answered through rigorous reasoning.

Parallel programming has been one of the main themes of theoretical computer science for decades. Existing parallel programming theories are thorough for specific kinds of parallelism, but heterogeneous parallelism often simultaneously involves several forms of parallelism including manycore parallelism without uniform cache coherence and large-scale clustering that relies on asynchronous message-passing to hide communication latencies.

This paper studies various parallelism in a unified programming model and reveals their essential commonality. As formalizing the entire dynamic behavior of all forms of parallelism is too demanding, we instead focus on patterns of data layout, distribution, transfer and their association with threads. These factors are often the most important aspects of compiler design and parallel-programming practice.

To achieve a performance-transparent level for programming, the representation must be expressive enough so as to convey enough source-level control information to the system level. Lack of expressiveness would force a programming tool to keep adding *ad hoc* commands for originally unforeseen application scenarios (see Section 6), or it deprives the system level of enough source information and must rely on an unpredictable optimizer to second-guess the programmer's intention. These approaches could be useful for some types of applications, but they do not offer the general solution that we are pursuing here.

Our approach instead extends traditional array types with locational information that describes how data are laid out in memory and distributed over multiple memory devices. Parray [9,28] is a parallel extension of C with generalised array types containing information about how the array elements are located in each memory device and distributed over multiple memory devices.

The focus of this paper is parallel programming for computing tasks with regular structures. Applications with irregular structures often have underlying regularity. For example a typical oil reservoir simulation has a largely three-dimensional mesh with a few irregular wells that connect some adjacent cells. Another common tactic is to use multiple regular mesh structures to represent an irregular mesh. Even for completely irregular computing tasks such as Breadth-First Search (BFS) of a large map graph, when the solution is mapped to a GPGPU, the regularity of the GPGPU structure becomes the main concern of optimisation.

Algebra have been used in parallelism for decades. Process algebra [26,11,30] focus on message-passing, while temporal logics [2,10] (which have equivalent algebraic representation [23]) are mostly used for shared-memory communications. Both forms of communication appear on a modern HPC system. For example, the communications among computing nodes include all kinds (synchronous, asynchronous, buffered or non-buffered) of message-passing, so does that of the network-on-chip of Kepler and Tilera and the PCI connection between each computing node and the manycore accelerators on it. Shared memory, on the other hand, appears on each computing node and among subset of cores. It is always possible to build a software-based virtual global address space for all memory devices. Such a protocol may be transparent for correctness, but whenever performance is concerned, the actual memory hierarchy becomes explicit to the programmer. Creative design of algorithms and programs is often necessary.

We follow the style of *algebraic semantics* [19,20]. The idea is to identify equational laws between programs and use program notation itself to define the meaning of programs. An equation goes either way, but if we stick to the convention of putting the transformed program on the left-hand side and the target program on the right-hand side, then such laws provide a transformation process and can form the foundation for compiler code generation. Algebraic techniques are introduced to describe index expressions generated from generalised array types. Such indexing is useful in computing address offsets of data elements as well as the thread/process id numbering involved in communications.

Section 2 introduces some simple examples of Parray programming. Section 3 describes the algebraic semantics of array types. Section 4 illustrates the versatile application of the algebraic techniques in code generation. In Section 5 the techniques are applied to a case study. Section 6 summarizes the related works.

2 Parallel Programming with Parray Types

Parray is a parallel extension to C that generates code for different manycore accelerators as well as multicore and cluster systems. Before studying the underlying theory, we first have a look at the syntax of an array-type example:

```
#parray {paged float[n][n]} A.                    (1)
```

The syntax resembles C macros and is compiled by the Parray preprocessor to generate various index expressions. When using Parray, the symbols for array types such as `A` are separate from the actual array objects (or pointers such as `a`). For example,

```
float* a;  #malloc A(a)
```

will use the information from the definition of `A` to allocate `n*n` floats in row-major `paged` memory (i.e. paging virtual memory) and assign the starting address to the pointer `a`. Just like C macros, the symbol `n` here can be a variable or an expression whose value is determined in runtime. The index expression `$A[i][j]$` generates a C expression `i*n+j` for in-scope indices `i` and `j`, and the corresponding element can be accessed by the expression `a[$A[i][j]$]`. An array type may take other shapes. For example, a type

```
#parray {[#A_1][#A_0]} B                          (2)
```

is the column-major variant of `A` in which the column dimension named `A_0` and the row dimension named `A_1` are swaped. We then have `$A[i][j]$` to coincide with `$B[j][i]$`. By checking the locality information of the type definitions, the Parray compiler is aware of all the information necessary for generating fast communication codes. For example, the command

```
#copy A(a) to B(b)
```

copies `n*n` data from address `a` to address `b` with columns and rows transposed.

Threads also form arrays. For example, a type

```
#parray {mpi[n]} P
```

indicates an array of `n` processes. A type may contain a mixture of thread dimensions and data dimensions:

```
#parray {[#P][#A_1]} C,
```

which describes a two-dimensional array distributed over `n` processes.

A more sophisticated type may have nested dimensions in a tree-like structure of sizes `(n*n)*m` with another memory type `dmem` for device memory on GPGPU accelerators and other dimension references in the tree:

```
#parray {dmem float[[n][n#B]#A_0][m]} D.          (3)
```

The **row sub-dimension** `D_0_0` of the column dimension `D_0` of `E` satisfies the equation: `$D_0_0[i]$ = $A_0[i*n]$*m` for in-scope indices `i`. The advantage of having a dimension tree is to keep a program mostly unchanged when a large dimension is further partitioned for deeper parallelisation and performance optimisation. **Further abstraction that makes the memory or thread types invisible is possible**, but the basic style of Parray programming assumes explicitness to allow control of data locality directly.

3 Syntax and Semantics

To study the most basic mathematical properties of Parray types, for now, we focus on dimension trees and their role in generating index expressions. Memory and thread types will be investigated in Section 4. In this paper the terms "array type" and "dimension" are interchangeable.

3.1 Syntax

The syntax of dimension trees is as follows:

$$\texttt{T ::= n | disp(n) | [T][T] | T\_h | T\#T | abs(T).} \tag{4}$$

Any positively integer-valued expression `n` can be the size of a dimension; a displacement `disp(n)` describes a dimension of size 1 with constant offset `n`; we consider two-dimensional constructor `[T][T]` of two sub-dimensions (multi-dimensions allowable as a simple syntactical extension); `T_0` or `T_1` indicates the column or row sub-dimension of `T`, respectively; type reference `T#T` describes a dimension whose size follows the left and indexing follows the right; the operator `abs(T)` forces a dimension's entire index expressions to be independent of its position in a larger dimension tree, respectively. We assume that **every type defined with `#parray` is absolute**. The real Parray specification also allows an arbitrary function `func(x) exp` as a user-defined index-expression macro. Since the definition (4) is already general, `func` is rarely needed in practice. All discussions in this paper are also applicable to multiple (more-than-two) dimensions at each level of a dimension tree.

We use `T[`α`]` to denote the index expression where each index α can be an integer expression `k` or further nested with sub-dimensions like `[`α_0`][`α_1`]`. We may simply write `T[i][j]` instead of `T[[i][j]]`. Not that if `T` is one-dimensional, then `T[i][j]`, `T_0` and `T_1` are syntactically invalid. For example, `D[[i][j]][k]` is a valid index expression, but `A_0[i][j]` is not.

For convenience, we simply write `A[k]` for the index expression `$A[k]$`. The equality between expressions is "`=`" instead of the C convention "`==`".

3.2 Some Ideas about Index Expressions

The semantic definitions of index expressions derive from common sense and programming intuition. For example, type `A` has a contiguous row dimension `A_1` and a discontiguous column dimension `A_0` with a constant gap `n` between adjacent offsets. We stipulate that

$$\texttt{A[k] = (k \% n*n),}\quad \texttt{A\_0[j] = (j \% n)*n,}\quad \text{and } \texttt{A\_1[i] = (i \% n).}$$

Each index is taken modulo against the dimension size to ensure the index to be in-scope, though the actual compiler **may not always generate the modulo operator** if the programmer is expected to manage the scoping. Note that the indexing of a relative dimension such as `A_0` has a constant gap of `n` positions

between adjacent indices. The gap is related to the dimension's location in the entire dimension tree. We also have `A[i][j] = A_0[i] + A_1[j]` for combined indices. A straightforward equation is identified:

$$\texttt{A[k] = A\_0[k/n] + A\_1[k\%n]}. \tag{5}$$

Unlike the **relative dimensions** of `A`, the dimensions of `B` are **absolute** on references to `A_0` and `A_1` such that `B_0[i] = A_1[i]` and `B_1[j] = A_0[j]`. The indexing of a absolute dimension does not depend on the sizes of other dimensions. The definition of `B[k]` follows the programmer's intuition:

$$\texttt{B[k] = B\_0[k/n] + B\_1[k\%n]}. \tag{6}$$

In general, a type like `D` may contain a mixture of relative and absolute dimensions. What is `D_0[i]` going to be? Again common sense prevails:

$$\texttt{D\_0[i] = A\_0[i/n*n] + B[i\%n]}.$$

From this example, we see that the semantics must keep track separately the relative (e.g. `i/n*n`) and absolute (e.g. `B[i%n]`) indexing of the sub-dimensions so that an external type reference only affects the relative part.

3.3 Dimension Sizes

The dimension sizes `size(T)` of array types satisfy some simple algebraic laws. The size of displacement is always 1; the size of a type reference `T#S` is that of the left-hand side; the operator `abs` does not change the dimension size.

Law 1. (1) `size(n) = n`
(2) `size(disp(n)) = 1`
(3) `size([T][S]) = size(T)*size(S)`
(4) `size([T][S]_0) = size(T)`
(5) `size([T][S]_1) = size(S)`
(6) `size((T#S)_h) = size(T_h)` `(h=0,1)`
(7) `size(abs(T)) = size(T)`

As a convention, `#A` is shorthand for `size(A)#A`.

3.4 Algebraic Semantics

The advantage of algebraic semantics is to use program notation itself to illustrate the meaning of the programs. Consider the special type `1` whose all indices are 0: `1[k] = (k%1) = 0`. Any dimension referring to `1` **loses its relative indexing and exposes its absolute indexing**. We shall use `|T|` to denote the relative part of a type. As the index expression `T[k]` provides both relative and absolute indexing while `T#1[k]` identifies only the absolute part, the relative part becomes their difference. We thus have the following defining decomposition:

$$\texttt{T[k] = |T|[k] + T\#1[k].}$$

The relative index expression of a dimension of size `n` is a modulo operator, while any displacement is added to the index. Both absolute parts are empty:

Law 2. (1) `|n|[k] = k%n`
(2) `|disp(n)|[k] = k+n`
(3) `n#1[k] = disp(n)#1[k] = 0`.

To compute the indexing of a two-dimensional array, a one-dimensional index `k` must be divided by the size of the row dimension to get the column sub-index. The relative column dimension is multiplied with the row size, while the absolute column dimension is not affected by the sizes of any other dimensions:

Law 3. (1) `|[T][S]|[k] = |T|[k/m]*m + |S|[k]`
(2) `([T][S]#1)[k] = T#1[k/m] + S#1[k]`

where `m = size(S)`.

The dimensions of a two-dimensional array are applied to two-dimensional indices respectively. Note that each α is either an index `k` or has two sub-dimensional indices $[\alpha_0][\alpha_1]$. Again, the relative column index expression is multiplied with the row size, while the absolute index expression is not:

Law 4. (1) $\texttt{|[T][S]|[[}\alpha_0\texttt{][}\alpha_1\texttt{]]} = \texttt{|T|[}\alpha_0\texttt{]*size(S) + |S|[}\alpha_1\texttt{]}$
(2) $\texttt{([T][S]\#1)[[}\alpha_0\texttt{][}\alpha_1\texttt{]]} = \texttt{T\#1[}\alpha_0\texttt{] + S\#1[}\alpha_1\texttt{]}$.

The laws of sub-dimensions `_h` reflect Law 4:

Law 5. (1) $\texttt{|[T][S]\_0|[}\alpha\texttt{]} = \texttt{|T|[}\alpha\texttt{]*size(S)}$
(2) $\texttt{|[T][S]\_1|[}\alpha\texttt{]} = \texttt{|S|[}\alpha\texttt{]}$
(3) $\texttt{([T][S]\_0\#1)[}\alpha\texttt{]} = \texttt{T\#1[}\alpha\texttt{]}$
(4) $\texttt{([T][S]\_1\#1)[}\alpha\texttt{]} = \texttt{S\#1[}\alpha\texttt{]}$.

The relative part of a type reference is the relative referred index expression applied to the relative referring index expression, while the absolute part is the absolute referring index expression added to the absolute referred index expression applied to the relative referring index expression. This indicates that the absolute indexing of the referring type is not affected by any further type reference:

Law 6. (1) $\texttt{|T\#S|[}\alpha\texttt{]} = \texttt{|S|[|T|[}\alpha\texttt{]]}$
(2) $\texttt{T\#S\#1[}\alpha\texttt{]} = \texttt{T\#1[}\alpha\texttt{] + S\#1[|T|[}\alpha\texttt{]]}$.

The operator `abs` forces the relative indexing to become absolute:

Law 7. (1) $\texttt{|abs(T)|[}\alpha\texttt{]} = \texttt{0}$
(2) $\texttt{abs(T)\#1[}\alpha\texttt{]} = \texttt{T[}\alpha\texttt{]}$.

3.5 Soundness and Completeness

The previous sub-section has laid out the axiomatic rules for index expressions. To study the consistency of the laws, it suffices to show that every type corresponds to a unique index expression. We first define the C-like syntax of normal-form integer expressions (with index `k`) as follows:

```
E(k) ::= n | k+n | k*n | k/n | k%n | E(k)+E(k) | E(E(k)).   (7)
```

Theorem 1 (Soundness). *The Law 2–7 uniquely transforms every array type's index expression into a normal-form integer expression.*

Proof. Induction on the syntactical complexity of of array types. Pure integer expressions without typed index expressions have 0 complexity. Arithmetic operators do not add complexity. The complexity of normal dimension `n` is assigned to 0. That of `disp(n)` is assigned to 1. The complexity of `|T|` and `T#1` is assumed to be equal to that of `T`. Two-dimensional constructor `[T][S]` and type reference `T#S` adds 1 to the max complexity between `T` and `S`. `abs(T)` adds complexity 1 to `T`. Then every Law 2–7 reduces the complexity from LHS to the RHS until reaching 0 free of typed index expressions. As the law applicable to each typed index expression is unique, the transformation is unique. □

We have focused on index expressions `T[k]` with one-dimensional indices. Two dimensional indices in `T[i][j]` are similar. In fact, not only every type corresponds to a unique normal-form integer expression, but also the design of Parray types ensures that every such integer expression is equal to the index expression of some array type (though not necessarily unique). The following laws show that every syntactical term in (7) has a Parray-type encoding:

Law 8. (1) `1#disp(n)[k] = n`
(2) `disp(n)[k] = k+n`
(3) `([disp(0)][n]_0)[k] = k*n`
(4) `([disp(0)][n#1])[k] = k/n`
(5) `n[k] = k%n`
(6) `([T][disp(0)#S])[k] = T[k]+S[k]`
(7) `|T|#S[k] = S[|T|[k]]`.

Theorem 2 (Completeness). *According to Law 2–7, for every pair of normal-form integer expressions* `E` *and* `E'`*, there exists a Parray type* `T` *such that*

`|T|[k]=E, T#1=E'` *and* `T[k]=E+E'`.

Proof. The LHS of Law 8(1)-(5) have empty absolute part. If both `T` and `S` have 0 as their absolute parts (i.e. `T#1[k]=0` and `S#1[k]=0`), so are `[T][disp(0)#S]` and `T#S`. That means any normal-form integer expression `E` corresponds to some array type `X` such that `|X|[k]=E` and `X#1[k]=0`. Thus any normal-form integer expression `E'` corresponds to some array type `Y` such that `|abs(Y)|[k]=0` and

`abs(Y)#1[k]=E'`. According to Law 8(6), `[X][disp(0)#Y]` is the array type satisfying the requirement. □

The soundness and completeness theorems have characterised the representational power of Parray types and ensure that more sophisticated array patterns become derivable from the basic notation.

This will help avoid the undesirable engineering practice(*e.g.* in HPF [29] and MPI [13]) that keeps adding fresh *ad hoc* notations (such as the CYCLIC distribution in HPF and various collectives in MPI) for unforeseen application scenarios. If deeply nested type references reduces readability, renaming with multiple type definitions easily simplifies it.

4 Versatile Applications of Algebraic Semantics

The algebra-semantic studies on array types also give rise to a variety of conditional-compilation techniques.

4.1 Contiguity

On real systems bandwidth often depends on the granularity of communication. For example, transferring data across the PCI between the main memory and the device memory of GPGPU requires a minimum of 8MB contiguous data block to achieve the peak bandwidth (between 3GB to 8GB). Discontiguous data transfers of smaller data segments will severely reduce the bandwidth. The following table indicates the estimated least segment size for peak bandwidth (granularity) of different channels.

	main mem	dev mem	shared mem	registers	infiniband
main mem	16KB	8MB			64KB
dev mem	8MB	64B	32B	32B	
shared mem		32B	4B	4B	
registers		32B	4B	4B	
infiniband	64KB				

We may use the compiler to generate code that checks (conservatively) the contiguity of any array type. Let the contiguity range expression `ran(T)` denote the contiguous range of a dimension and the step expression `stp(T)` denote the gap of offset between adjacent indices. If a dimension like `B` in (2) is discontiguous, the contiguity range is 0 and the step is arbitrary.

In a two-dimensional type, if the contiguity range of the row dimension fits the step of the column dimension, then the overall step equals that of the row dimension; the contiguity range is 0 for a discontiguous type.

Law 9. (1) `ran([T][S]) = ran(T) * size(S)`

(2) `stp([T][S]) = stp(S)`

if `ran(S)>=size(S)` *and* `stp(T) = size(S) * stp(S)`*; otherwise,* `ran([T][S])=0`.

The contiguity ranges or steps of sub-dimensions are identified as follows:

Law 10

(1) `ran([|T|][S]_0) = ran(T)` (2) `stp([|T|][S]_0) = stp(T)*size(S)`
(3) `ran([T#1][S]_0) = ran(T)` (4) `stp([T#1][S]_0) = stp(T)`
(5) `ran([T][S]_1) = ran(S)` (6) `stp([T][S]_1) = stp(S)`.

The contiguity range of a referring type is also the contiguity range of a type-reference composition if it is bounded by the contiguity range of the referred type:

Law 11. (1) `ran(T#S) = ran(T)`
(2) `stp(T#S) = stp(T)*stp(S)`

if `ran(T)*stp(T)<ran(S)`.

When the range of `S` is exceeded, the range and step become as follows:

Law 12. (1) `ran(|T|#S) = ran(S)/stp(|T|)`
(2) `stp(|T|#S) = stp(|T|)*stp(S)`
(3) `ran(T#1#S) = ran(T#1)`
(4) `stp(T#1#S) = stp(T#1)`

if `ran(|T|)*stp(|T|)>=ran(S)`.

The operator `abs` does not affect contiguity range and step:

Law 13. (1) `ran(abs(T)) = ran(T)`
(2) `stp(abs(T)) = stp(T)`.

The following theorem shows that the index expression `T[k]` is linear with respect to the indices within the contiguity range `ran(T)`, though it is not true vice versa. That means the linearity checking is conservative.

Theorem 3 (Effectiveness I). *Law 9-13 uniquely transform the contiguity range and step expressions to normal-form integer expressions. The contiguity range and step expressions satisfy that for all indices* `k<ran(T)`*, we have* `T[k]=k*m+T[0]`.

Proof. The proof of the first half uses the same technique of Theorem 1. By routinely checking Law 2-7 and Law 9-13. □

A dimension `T` is "**contiguous**" if the contiguity range `ran(T)` is no-smaller-than `size(T)` and the step is `1`. The contiguity boolean expression is defined:

`cnt(T) = (ran(T)>=size(T) && stp(T)=1)`.

Data movement between arrays of contiguous types can be performed as a single large data block or message.

4.2 Range Estimation

In practice, it is often useful to estimate the range of a dimension's index expression. For example, when duplicating data from the host's main memory to GPGPU accelerator's device memory via PCI, it is possible to adjust the order of the elements so that the GPGPU may use a different memory layout for the same array to better coalesce memory accesses. As PCI's bandwidth requires nearly 8MB granularity to achieve the peak bandwidth, such order adjustment cannot be performed during PCI transfer. If the estimated **size of the memory region** (potentially with small gaps) does not exceed the size of the array, an effective approach is to copy the entire region (or as segments, depending on the particular strategy) to a buffer on GPGPU and call a GPGPU kernel to re-copy the elements to the destination addresses. Order adjustment within a GPGPU's device memory can benefit from its high memory bandwidth.

From the the syntax of normal-form inter expressions (7), it is obvious that `T[0]` is always the starting offset of the memory layout, but the modulo operator complicates the estimation of the **upper bound** (denoted as `sup(T)`). Algebraic laws again become helpful in this regard.

Law 14. (1) `sup(n) = n-1`
(2) `sup(disp(n)) =` ∞

The upper bound of a relative column dimension is multiplied with the row size, but that of the absolute column dimension is not:

Law 15. (1) `sup([|T|][S]) = sup(|T|)*size(S)+sup(S)`
(2) `sup([T#1][S]) = sup(T#1)+sup(S)`.

The same rule applies to sub-dimensions:

Law 16. (1) `sup([|T|][S]_0) = sup(|T|)*size(S)`
(2) `sup([T#1][S]_0) = sup(T#1)`
(3) `sup([T][S]_1) = sup(S)`.

The upper bound of a type reference is that of the referred type. The operator `abs` does not affect the upper bound:

Law 17. (1) `sup(T#S) = sup(S)`
(2) `sup(abs(T)) = sup(T)`.

The following theorem states the range of an index expression.

Theorem 4 (Effectiveness II). *Law 14-17 uniquely transform the contiguity range and step expressions to normal-form integer expressions. For any* `T` *and index* `k`*, we have* `T[k]>=T[0]` *and* `T[k]<=sup(T)`.

Proof. By routinely checking Law 2-7 and Law 14-17. □

We shall use `dense(T)` to denote the boolean expression `sup(T)<=size(T)`.

4.3 Sub-programming and Conditional Compilation

In Parray, array types can be passed as parameters to a sub-program, which works very much like a general C macro or compile-time lambda calculus. The following Parray code declares a sub-program `DataTransfer` (**sketch**) that implements the command `#copy SRC(src) to DST(dst)` in which `SRC` and `DST` are type arguments.

```
#subprog DataTransfer(dst, DST, src, SRC)
    #paif (cnt(SRC) && cnt(DST)) {
        #insert ContTransfer(dst, DST, src, SRC) {}}
    #elseif (SRC:pinned && DST:dmem && dense(SRC)) {
        #parray {dmem elm(DST)[size(DST)] BUF
        {#create BUF(buf)
        #insert ContTransfer(buf, BUF, src, pinned BUF) {}
        #insert DMEMTransfer(dst, DST, buf, dmem SRC) {}
        #destroy BUF(buf)}}
    #else {...}
#end
```

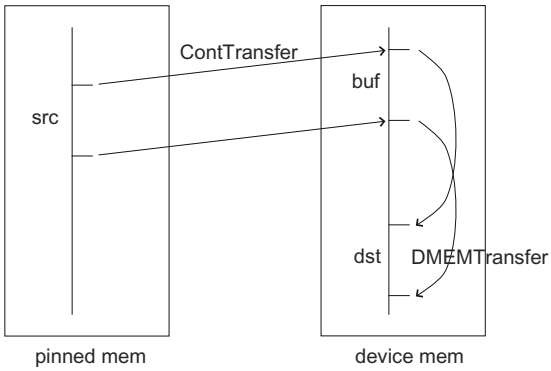
The conditional-compilation command `#paif..#elseif..#else..` checks the patterns of the input types and generate optimised code according to the type information. If both `SRC` and `DST` are contiguous, then the sub-program `ContTransfer` for contiguous data transfer is inserted. If the source array type and the destination type are `paged`, the contiguous data transfer between them becomes a single `memcpy` command; the data transfer from `pinned` memory to `dmem` on GPGPU, on the other hand, generates a `cudaMemcpy` command.

```
#subprog ContTransfer(dst, DST, src, SRC)
    #paif (SRC:paged && DST:paged) {memcpy(...);}
    #elseif (SRC<=pinned && DST<=dmem) {
        cudaMemcpy(...,cudaMemcpyHostToDevice);}
    ......
#end
```

If the data transfer from `pinned` memory to `dmem` is discontiguous and `SRC` is dense, then an image array `buf` of `SRC` is created in device memory. The data are first copied to the image buffer and then transferred to the destination using `DMEMTransfer`, which is a one-liner Parray code that performs any data movement within GPGPU device memory.

```
#subprog DMEMTransfer(dst, DST, src, SRC)
    #forall cuda[size(SRC)](dst,src) as (y,x) {
        y[DST[tid]] = x[SRC[tid]];}
#end
```

The code creates an array of `size(SRC)` GPGPU threads, each copies one element from the position `SRC[tid]` of array `src` to the position `DST[tid]` of array `dst`.



Note that this implementation is only a sketch of the actual Parray system library, but from the examples, we get a glimpse of how the memory-type information helps a sub-program in the system library to generate hardware-related code on behalf of the compiler. This approach of using sub-programs for code generation is particularly convenient as new manycore hardware devices and models are constantly introduced in every few months.

5 Case Study: Large-Scale FFT

For *small-scale* FFTs whose data are held entirely on a GPU device, their computation benefits from the high device-memory bandwidth [1]. This conforms to an application scenario where the main data are located on dmem, and FFT is performed many times. Then the overheads of PCI transfers between hmem and dmem are overwhelmed by the computation time.

If the data size is too large for a GPU device or must be transferred from/to dmem every time that FFT is performed, then the PCI bandwidth becomes a bottleneck. The time to compute FFT on a GPU will likely be overwhelmed by data transfers via PCIs. This is the scenario for large-scale FFTs on a GPU cluster where all the data are moved around the entire cluster and between hmem and dmem on every node. The performance bottleneck for a GPU cluster will likely be either the PCI between hmem and dmem or the network between nodes — whichever has the narrower bandwidth.

In our previous work [8], we proposed an FFT algorithm called PKUFFT for GPU clusters. The original implementation uses CUDA, Pthread, MPI and even the low-level infiniband library IB/verbs for performance optimization. That implementation is unportable and specific to a 16-node cluster with dual infiniband cards and dual Tesla C1060 GPUs on each node. To port that code to a large CPU-GPU cluster Tianhe-1A, we first rewrite the code in Parray and then recompile it on the target machine, drastically reducing its length (from 400 lines to 30 lines) while preserving the same depth of optimization. The code has been deployed to support large-scale turbulence simulation.

Our 3D FFT algorithm first distributes a 3D array with dimensions Z, Y and X along (the left-most) dimension Z. Every computing node holds N/P 2D planes for Y and X dimensions (in which X is contiguous). Every 2D plane is transferred to the GPU for 2D FFT computation (using the existing library CUFFT) and then transferred back to the main memory. All computing nodes then perform an Alltoall-like collective communication to aggregate the Z dimension on each computing node and redistribute the array along dimension Y. Data are then computed for 1D FFT ad sent back to the main memory.

A major operation of the algorithm requires transposing the entire array, which usually involves main-memory transposition within every node and Alltoall cluster communication. The main optimization of the algorithm [8] is to re-arrange and decompose the operation into small-scale GPU-accelerated transposition, large-scale Alltoall communication and middle-scale data-displacement adjustment that is performed during communications. Then the main-memory transposition is no longer needed! The price paid is to use a non-standard Alltoall with discontiguous process-to-process communications.

```
#parray {mpi[K]} L
#detour L{
 #parray {pinned float2 [N/K][[K][N/K]][N]} G
 #parray {pinned float2 [disp i][N][N]} F
 #parray {dmem float2 [N][N]} Q
 #parray {dmem float2 [#Q_1][#Q_0]} R
 #parray {[[#L][#G_0][#G_1_0][#G_1_1]][#G_2]} S
 #parray {[[#G_1_0][#G_1_1][#L][#G_0]][#G_2]} T
 float2* g; #create G(g)
 float2* gbuf; #create G(gbuf)
 float2* q; #create Q(q)
 float2* qbuf; #create Q(qbuf)
 cufftHandle plan2d;
 cufftPlan2d(&plan2d,N,N,CUFFT_C2C);
 for(int i=0; i<N/K; i++) {
  #insert DataTransfer(q,Q, g,F){}
  cufftExecC2C(plan2d,q,q,CUFFT_FORWARD);
  #insert DataTransfer(gbuf,F, q,Q){}
 }
 #insert DataTransfer(g,T, gbuf,S){}
 cufftHandle plan1d;
 cufftPlan1d(&plan1d,N,CUFFT_C2C,N);
 for(int i=0; i<N/K; i++) {
  #insert DataTransfer(q,Q, g,F){}
  #insert DataTransfer(qbuf,R, q,Q){}
  cufftExecC2C(plan1d,qbuf,qbuf,CUFFT_FORWARD);
  #insert DataTransfer(q,Q, qbuf,R){}
  #insert DataTransfer(gbuf,F, q,Q){}
 }
```

```
 #insert DataTransfer(g,S, gbuf,T){}
 cufftDestroy(plan2d);
 cufftDestroy(plan1d);
 #destroy G(g) #destroy G(gbuf)
 #destroy Q(q) #destroy Q(qbuf)
}
```

This code consists of a series of data-transfer operations that we already studied in previous sections. The main 3D complex data are stored in array `g` of type `G`. Another array `gbuf` acts as a buffer. The inner dimension `G_2` is contiguous; `G_1` is the middle dimension; the outer dimension is a combination of thread dimension `L` and `G_0`. Each MPI processes in `L` contains `N/K` pages of size `N*N`. In the first step, every page (with middle and inner dimensions) is transferred to the dmem array `q` for 2D FFT computation (by calling CUDA library) with results transferred back into `qbuf`. The following communication over the entire network is the non-standard discontiguous Alltoall communication pattern. The communication effectively swaps the outer and middle dimensions, so that the middle dimension is aggregated on each MPI process. Every 2D page of the middle and inner dimensions is transferred to dmem again. Before performing batched 1D FFT on the new middle dimension, we use GPU transposition to swap the middle and inner dimensions to make the middle dimension contiguous. The original positions of the data are restored after FFT by GPU transposition and communication.

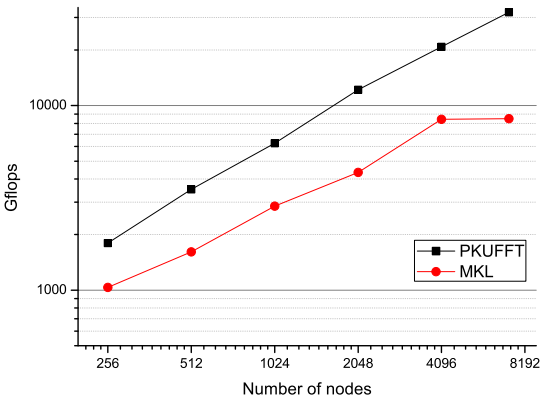


Fig. 1. PKUFFT *vs.* Intel Cluster MKL on Tianhe-1A

The above FFT code is tested on Tianhe-1A using up to 7168 nodes, each with 24GB main memory, two 6-core Intel Processors and one Tesla Fermi 448-core GPU. The special customized network has 80Gb/s bandwidth for each node and a fat tree structure for switching. CUDA version is 3.0; CUFFT version is 3.1. For comparison, Intel Cluster MKL (or CMKL) 10.3.1.048 is used on the same cluster but does not use GPU. CMKL is already highly optimized because of the heavy

communication load of very-large FFTs. The tests are performed for 3D FFT of different sizes for single-precision C2C forward (with returning communication that restores the data to their original positions). Double-precision FFTs perform at the half speed of single precision (on Fermi as well as data transfers). GPU-accelerated FFT also scales better than CPU-based FFT. We mainly develop R2C and C2R single-precision 3D FFTs, which are used for large direct numerical simulation of turbulent flows up to the scale of 14336 3D on Tianhe-1A. .

6 Literature Survey

The programming interfaces considered for comparisons include Chapel [5], Co-Array Fortran (CAF) [27], HPF [29] HMPP [15], Hierarchically Tiled Arrays (HTA) [16], Titanium [31], Stanford PPL [3], UPC [32], X10 [7,17], ZPL [6,12], Global Arrays [22] and Sequoia [14].

Among the existing language designs, HTA is perhaps the closest. The dimension tree, type references, and thread arrays are not supported by HTA. The array representation in Parray is more expressive. The theory part shows the algebraic completeness of the representation. Hierarchical tiles arrays assume several default levels (for multicore/cluster parallelism). Parray's dimension trees are logical and not tied to a specific memory structure.

A large class of languages are PGAS languages. If there exists a well-optimised PGAS library such as Global Arrays (`gamem`), Parray can generate code to invoke that library. The programmer needs to specify the memory type explicitly to calls the communication library.

Arrays and the associated operations are considered in category theory in [25], but data layout in memory and distribution over different memory devices are not on their agenda. In functional programming and type theory [18,24,4] as well as works centered on APL [21] the focus is the types and shapes of operations on arrays. Array partitioning is considered for the operations. Memory-related data layout and parallelism-related distribution are not considered.

7 Conclusion

The reality of heterogeneous parallel programming indicates that using one simple programming notation or a single theory to handle all forms of parallelism is unrealistic. A more realistic view is to introduce a number of programming tools.

The current engineering solution by the manufacturers is to provide a variety of tools for different hardware devices, *e.g.* Pthread/OpenMP for multicore, CUDA for manycore and MPI for clustering. We would like to see a change from this hardware-based distinction into more application-based distinction for programming tools. For example, the programmer can choose a programming tool to convert problems such as finite elements with irregular structures into a regular structure, a different tool to handle runtime scheduling of computing

tasks in the entire system, or a tool like Parray that is specialized in handling regularity in numerical applications.

Parray only provides abstraction for regular data structures like arrays. Irregular data structures such as trees and graphs must be encoded as arrays to benefit from Parray's integrated code generation. The encoding is left to the user or any higher-level software layers/libraries. Parray's performance transparency makes sure that any higher-level layers implemented on top of Parray will not be performance-wise penalized because of using Parray instead of the low-generated-code-level libraries.

The mission of this paper is to convert the engineering challenges of programming emerging architecture into the realm of programming theories. There are a number of ways to generalize the formal models. For example, the type reference as well as sub-programming may allow recursion and fixpoints. It will be interesting to see whether further exploration in these theoretical directions lead to results that are relevant in practice.

The design of Parray is also influenced by the desire to have a complete representation. Such unprecedented expressiveness and completeness compared to various previous array notations is crucial: it makes sure that 1. a wide range of heterogenous parallel applications are supported; 2. it offers an easy-to-understand intuition about what are and are not representable with the array types; and 3. it forms a flexible interface between the programmer and the hardware system and conveys enough source-level information to the implementation level for performance optimization.

References

1. CUDA CUFFT Library, Version 2.3. NVIDIA Corp. (2009)
2. Abadi, M., Lamport, L.: Conjoining specifications. ACM Trans. Program. Lang. Syst. 17(3), 507–534 (1995)
3. Chafi, H., et al.: A domain-specific approach to heterogeneous parallelism. In: PPoPP 2011 (2011)
4. Chakravarty: Accelerating Haskell array codes with multicore gpus. In: Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming, DAMP 2011, pp. 3–14 (2011)
5. Chamberlain, B., Callahan, D., Zima, H.P.: Parallel programmability and the Chapel language. IJHPCA 21(3), 291–312 (2007)
6. Chamberlain, B., et al.: The high-level parallel language ZPL improves productivity and performance. In: IJHPCA 2004 (2004)
7. Charles, P., et al.: X10: An object-oriented approach to nonuniform cluster computing. In: OOPSLA 2005 (2005)
8. Chen, Y., Cui, X., Mei, H.: Large-scale FFT on GPU clusters. In: ACM Inter. Conf. on Supercomputing, ICS 2010, pp. 50–59 (2010)
9. Chen, Y., Cui, X., Mei, H.: Parray: A unifying array representation for heterogeneous parallelism. In: PPoPP 2012, pp. 171–180 (2012)
10. Chen, Y., Sanders, J.: Logic of global synchrony. ACM Transactions on Programming Languages and Systems 26(2), 221–262 (2004)
11. Cleaveland, R., Luettgen, G., Natarajan, V.: Priority and abstraction in process algebra. Information and Computation 205(9), 1426–1428 (2007)

12. Deitz: The design and implementation of a parallel array operator for the arbitrary remapping of data. In: PPoPP 2003, pp. 155–166 (2003)
13. Dongarra, J., Snir, M., Otto, S., Walker, D.: A message passing standard for MPP and workstations. Communications of the ACM 39(7), 84–90 (1996)
14. Park, J.-Y., et al.: A portable runtime interface for multi-level memory hierarchies. In: PPoPP 2008, pp. 143–152 (2008)
15. Francois, B.: Incremental migration of C and Fortran applications to GPGPU using HMPP. Technical report, HIPEAC (2010)
16. Ganesh, B., et al.: Programming for parallelism and locality with hierarchically tiled arrays. In: PPoPP 2006, pp. 48–57 (2006)
17. Grothoff, C., Palsberg, J., Saraswat, V.: A type system for distributed arrays (2012) (preprint)
18. Hains, G., Mullin, L.M.R.: Parallel functional programming with arrays. Comput. J. 36(3), 238–245 (1993)
19. Hoare, C.A.R., et al.: Laws of programming. Communications of the ACM 30(8), 672–686 (1987)
20. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall (1998)
21. Iverson, K.E.: Operators. ACM Trans. Program. Lang. Syst. 1(2), 161–176 (1979)
22. Nieplocha, J.J., Harrison, R.J., Littlefield, R.J.: Global arrays: A nonuniform memory access programming model for high-performance computers. The Journal of Supercomputing 10(2) (1996)
23. Karger, B.V.: Temporal algebra. Mathematical Structures in Computer Science, 32–80 (1996)
24. Keller, G., Chakravarty, M.M., Leshchinskiy, R., Peyton Jones, S., Lippmeier, B.: Regular, shape-polymorphic, parallel arrays in Haskell. In: ICFP 2010, pp. 261–272 (2010)
25. Macedo, H.D., Oliveira, J.N.: Matrices as arrows! In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 271–287. Springer, Heidelberg (2010)
26. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
27. Numerich, R., Reid, J.: Co-Array Fortran for parallel programming. SIGPLAN Fortran Forum 17(2), 1–31 (1998)
28. PKU Manycore Software Research Group. Parray user's manual (2013), `http://code.google.com/p/parray-programming/`
29. Richardson, H.: High Performance Fortran: history, overview and current developments. Technical Report TMC-261, Thinking Machines Corporation (1996)
30. Sangiorgi, D., Walker, D.: The pi-calculus: a Theory of Mobile Processes. Cambridge Universtity Press (2001)
31. Yelick, K., et al.: Titanium: A high-performance Java dialect. In: ACM, pp. 10–11 (1998)
32. Zheng, Y., et al.: Extending Unified Parallel C for GPU computing. In: SIAM Conf. on Parallel Processing for Scientific Computing (2010)

Modeling and Specification of Real-Time Interfaces with UTP

Hung Dang Van and Hoang Truong*

University of Engineering and Technology
Vietnam National University, Hanoi, Vietnam
{dvh,hoangta}@vnu.edu.vn

Abstract. Interface modeling and specification are central issues of component-based software engineering. How a component will be used is specified in its interface. Real-time interfaces are interfaces with timing constraints relating the time of outputs with the time of inputs. The timing constraint of an interface may depend on the resource availability for the component. In this paper, we propose a general model for real-time interfaces. At a time during execution, an interface behaves according to a contract made with environment about its functionality as well as execution time to fulfill the contract. This contract is specified as a timed design using the UTP notations, and depends on the computation histories of the interface. We model this dependence as a partial function from computation histories of the interface to real-time contracts. How interfaces are composed to form new interfaces, how interfaces are refined, and how to represent interfaces finitely are also considered in this paper. We show that checking the consistency between an environment and an interface and checking the refinement between two interfaces when they are represented by an automaton can be done effectively.

1 Introduction

Component-based design has been an efficient divide-and-conquer technique for the development of complex real-time embedded systems. A key role for this technique is interface modeling and specification. There has been a lot of significant progress towards a comprehensive theory for interfaces, see for example [1–6]. In those works different aspects of interfaces have been modeled and specified: interaction protocols, contracts, concurrency, timing, input-output relations, synchrony and asynchrony, etc. However, none among them can integrate all those aspects which help to analyze the relation between different aspects of the interfaces and find out conditions on which different aspects can be separated to simplify the analysis and verification. For untimed component interfaces, a recent work by Tripakis [7] has proposed a nice and comprehensive theory.

In this work, we extend that theory for real-time interfaces with the use of UTP developed by He and Hoare [8]. In this framework, we consider a component

* This work was supported by the research Project QG.11.29, Vietnam National University, Hanoi.

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 136–150, 2013.

as to provide a set of services, each service is specified by a "timed design". A timed design consists of a precondition on the input variables, and a relation between input variables and output variables together with the time constraint for the availability of the output. This time constraint is of the form $c \leq \ell \leq d$ where c is the best case execution time and d is the worst case execution time to fulfill the computation, and ℓ is a special temporal variable representing excution time. Which service is offered at a time depends on the current state of the component. The current state of the component expresses much more information than a protocol. From the current state, we can see not only how the environment interacted with the component so far, but also how the component has evolved. A state is a timed sequence of rounds, each round consists of an input value together with the time the input is given, and output value responding to that input value together with the actual execution time. So, the set of all states of a component is infinite. We then consider how an interface interacts with its environment, and derive the set of all reachable real-time behaviors for the system which can help to estimate the worst case execution time and to analyze the schedulability for multithreaded environments. We use the plugability as the condition for interface refinement and find that this condition can also be verified syntactically. In practice, an interface can only provide a finite number of services, and since a service is related with a subset of states, we can introduce an equivalence relation to partition the set of states into a finite number of subsets and use a labeled automaton to represent an interface. With that kind of representation, we can easily perform some operations on interfaces like interface compositions and refinement, and the verification of some properties. Note that when timing constraints are ignored, and specified as the interval $[0, 0]$ our model becomes an untimed synchronous interface one.

The paper is organized as follows. The next section presents our general real-time interface model. Section 3 considers some operations on interfaces. The representation of interfaces as automata and checking the plugability is presented in Sec 4. Related work and discussion is presented in Section 5. The last section is the conclusion of our paper.

2 Timed Relational Interface

A static service provided by a real-time component C can be specified as a contract on a signature (X, Y), where X and Y respectively are disjoint sets of input and output variables of the component. The contract can be represented by a so-called "timed design" which was introduced by us in [3] as a timed extension of a design developed by He and Hoare [8]. Let ℓ be a specific temporal variable for execution time, $\ell \notin X \cup Y$.

Definition 1. *(Timed design)*

1. *A timed design on signature* (X, Y) *is of the form:*

$$\rho = p \vdash (R, c \leq \ell \leq d)$$

where p is a predicate on variables in X called precondition of ρ, R is a relation from X to Y called post-condition of ρ, and c, d are non-negative real numbers satisfying $c \leq d$. Let us denote R as ρ_f and $c \leq \ell \leq d$ as ρ_t for timed design ρ. We call ρ_t the time constraint of ρ[1]. Timed design ρ simply says that if it is called with the value of inputs satisfying p then it will terminate and give values for output variables after at least c time units and at most d time units, and R is satisfied at the termination.

2. *For timed designs $\rho = p \vdash (R, c \leq \ell \leq d)$ and $\rho' = p' \vdash (R', c' \leq \ell \leq d')$, ρ is said to be a refinement of ρ', denoted as $\rho \sqsubseteq \rho'$ iff $p' \Rightarrow p$, $R \Rightarrow R'$ and $[c, d] \subseteq [c', d']$. When $\rho \sqsubseteq \rho'$ and $\rho' \sqsubseteq \rho$ we say ρ and ρ' are equivalent.*

Let $\mathbb{R}^+$ denote the set of non-negative real numbers. A computation round (or simply a round) is a pair $(\mathcal{V}, I)$, where $\mathcal{V}$ is a value assignment for variables in $X \cup Y$, and I is a time interval $[b, e]$, $b, e \in \mathbb{R}^+$, $b \leq e$. A round $r = (\mathcal{V}, I)$ is said to satisfy a timed design $\rho = p \vdash (R, c \leq \ell \leq d)$, denoted as $(\mathcal{V}, I) \models \rho$, iff $\mathcal{V}|_X \models p$, $\mathcal{V} \models R$ and $c \leq length(I) \leq d$, where $\mathcal{V}|_X$ is the restriction of $\mathcal{V}$ on the set of variables X, and $length(I)$ is the length of interval I. So, when $p \equiv false$, no round can satisfy ρ. In this definition of satisfiability, only the length of the interval of a round (not the interval itself) and the value assignment play role. Therefore, we says that rounds $(\mathcal{V}, I)$ and $(\mathcal{V}, I')$ are equivalent if $length(I) = length(I')$. For any equivalent rounds r, r' and a timed design ρ it holds that $r \models \rho$ if and only if $r' \models \rho$. A round $r = (\mathcal{V}, [b, e])$ is said to be before (right before) a round $r' = (\mathcal{V}', [b', e'])$, or equivalently, r' is after r, if $e \leq b'$ (resp. $e = b'$). A sequence of rounds $r_1 r_2 \dots r_n$ such that r_{i+1} is after r_i for all $1 \leq i < n$ is called a state. A sequence of consecutive rounds is a state $r_1 r_2 \dots r_n$ such that r_{i+1} is right after r_i for all $1 \leq i < n$.

Let $\mathcal{S}(X, Y)$ denote the set of all states, $\mathcal{D}(X, Y)$ denote the set of all timed designs over signature (X, Y). We give the following definition to real-time interfaces.

Definition 2. *(Real-time Interface) A real-time interface is a tuple $\mathcal{I} = (X, Y, \xi)$, where (X, Y) is a signature, ξ is a partial function from $\mathcal{S}(X, Y)$ to $\mathcal{D}(X, Y)$ satisfying:*

- *$\xi(\epsilon) = \rho_0$ is defined, where ϵ is the empty sequence.*
- *If $\xi(r_1 r_2 \dots r_n)$ is defined, then $\xi(r_1 r_2 \dots r_{n-1}) = \rho_{n-1}$ is also defined, and $r_n \models \rho_{n-1}$.*

When $\xi(r_1 r_2 \dots r_n)$ is defined, we call $r_1 r_2 \dots r_n$ a reachable state of (X, Y, ξ).

So, the set of all reachable states of a real-time interface $\mathcal{I} = (X, Y, \xi)$ forms a prefix-closed subset of $\mathcal{S}(X, Y)$. A reachable state s for which there is no round r such that sr is reachable is called a deadlock state of $\mathcal{I}$. Deadlock states are those reachable states that cannot be expanded. There are two reasons that a given

[1] ρ could be written as a formula $p \vdash \rho_f \wedge \rho_t$. However, for the purpose of this paper it is more convenient to write ρ as $p \vdash (\rho_f, \rho_t)$.

state s is not expandable: either the function ξ is undefined for any extension of the state s, or the timed design $\xi(s)$ is not satisfiable. In the sequence, we will assume that for any state s, if $\xi(s)$ is defined then it is a satisfiable timed design. Hence, deadlock states are those reachable states for which the function ξ is not defined for any extension of them.

Example 1. Consider an interface that takes x as its unique input, and outputs y as the average of the values of x that it has received so far. Only the values of x that are in between two thresholds low and $high$ are permitted to calculate the average. The interface is specified as $(\{x\}, \{y\}, \xi)$, where $\xi((\mathcal{V}_1, I_1) \dots (\mathcal{V}_n, I_n)) = ((x \leq high \wedge low \leq x) \vdash ((\sum_{j=1}^{n} \mathcal{V}_j(x) + x)/(n+1) = y, 0 \leq \ell \leq 1))$ for any state $(\mathcal{V}_1, I_1) \dots (\mathcal{V}_n, I_n)$ such that $\mathcal{V}_i(x) \in [low, high]$, $\mathcal{V}_i(y) = \sum_{j=1}^{i} \mathcal{V}_j(x)$ and $length(I_i) \leq 1$.

There are two interesting properties for this interface: it does not have deadlock state, and the range of the function ξ is infinite.

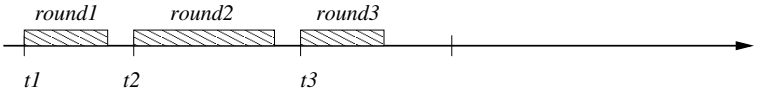


Fig. 1. A state as a timed execution of an interface

Let $\mathcal{R}(\mathcal{I})$ denote the set of reachable states of interface $\mathcal{I}$. In a reachable state, the time interval of a round represents an execution of a service provided by the component of the interface. When the environment invokes a service, it starts a round. So, it is the environment who decides the time to start a round, and the component decides when a round ends. Therefore, it is more practical, if for two states $s = r_1 \dots r_k$ and $s' = r'_1 \dots r'_k$ for which r_i and r'_i are equivalent for all $1 \leq i \leq k$, we have $\xi(s) = \xi(s')$. This gives right to the following definition. We say two states $s = r_1 \dots r_k$ and $s' = r'_1 \dots r'_n$ are equivalent iff $k = n$ and for all $1 \leq i \leq k$ the rounds r_i and r'_i are equivalent.

Definition 3. *(Timeless Interface) A real-time interface (X, Y, ξ) is timeless (or input-time independent) iff $\xi(s) = \xi(s')$ for any reachable equivalent states s, s'. (X, Y, ξ) is completely timeless (or input/output-time independent) iff $\xi(s) = \xi(s')$ for any reachable states $s = (\mathcal{V}_1, I_1) \dots (\mathcal{V}_n, I_n)$ and $s' = (\mathcal{V}_1, I'_1) \dots (\mathcal{V}_n, I'_n)$*

So, the real-time services provided by an interface do not depend on the time of inputs if the interface is timeless, and do not depend on the time of inputs and outputs if the interface is completely timeless. From now on in this paper, we consider only timeless interfaces, and the reachable states with consecutive rounds will be selected to be the representatives of equivalence classes. Those representatives are states that represent the busiest behavior of components, and are used to analyze the real-time capacity of interfaces like for which arrival rate and which deadline can be met for the worst case.

The range of ξ in an interface $\mathcal{I}$ could be infinite like in the example given above, and this makes it difficult to represent interfaces. As we have mentioned in the introduction, in practice, a component can only provide a finite number of services. The interface of this kind of components is called finite.

Definition 4. *A real-time interface* (X, Y, ξ) *is*

- *finite iff range*(ξ) *is finite,*
- *stateless iff range*(ξ) *has only one element, i.e.* ξ *is a constant mapping.*

Now we give a definition of interface environments and study how an interface will be used by an environment. Let us define a timed behavior to be a sequence $(\mathcal{V}_1, t_1)(\mathcal{V}_2, t_2) \dots (\mathcal{V}_m, t_m)$, where $\mathcal{V}_i$ is a value assignment for variables in $X \cup Y$, $t_i \in \mathbb{R}^+$, $t_i \leq t_{i+1}$, $i = 1, \dots, m-1$, $t_1 = 0$. Let $\mathcal{W}(X, Y)$ be the set of all timed behaviors over (X, Y). For a set of variables V, let $\mathcal{F}(V)$ denote the set of all predicates with free variables in V.

Definition 5. *(Environment) An environment over the signature* (X, Y) *is a tuple* $E = (X, Y, h)$*, where* h *is a partial function from* $\mathcal{W}(X, Y)$ *to* $\mathcal{D}(X, Y)$*, respectively, satisfying:*

- $h(\epsilon) = f_0 \vdash (g_0, l_0 \leq \ell \leq u_0)$ *is always defined.*
- *If* $h(w_1 w_2 \dots w_n) = f_n \vdash (g_n, l_n \leq \ell \leq u_n)$ *is defined then*

$$h(w_1 w_2 \dots w_{n-1}) = f_{n-1} \vdash (g_{n-1}, l_{n-1} \leq \ell \leq u_{n-1})$$

  is also defined for which $\mathcal{V}_n \models f_{n-1} \wedge g_{n-1}$ *and* $t_n - t_{n-1} \geq u_{n-1}$ *hold, where* $w_i = (\mathcal{V}_i, t_i)$*,* $i \leq n$*.*

When $h(w_1 w_2 \dots w_n)$ *is defined, we call* $w_1 w_2 \dots w_n$ *a reachable timed behavior of environment* E*. Let* $\mathcal{B}(E)$ *denote the set of all reachable timed behaviors of environment* E*.*

As for the function ξ *of interfaces, we assume also that* $h(w)$ *is satisfiable whenever it is defined.*

Note that an environment $E = (X, Y, h)$ is not an interface since h is a partial function on $\mathcal{W}(X, Y)$, while in an interface $\mathcal{I} = (X, Y, \xi)$, the partial function ξ is on $\mathcal{S}(X, Y)$. Intuitively, a reachable timed behavior $w_1 w_2 \dots w_n$ of E represents an interaction between E and an interface. For any $i = 1, \dots, n$, at time t_i, E issues an input X that satisfies f_{i-1} to the interface, and expects an output Y at some time in the time interval $[t_i + l_{i-1}, t_i + u_{i-1}]$ from the interface, and that output is related with the issued input by relation g_{i-1}. This is described by $\mathcal{V}_i \models f_{i-1} \wedge g_{i-1}$. The time t_{i+1} at which it issues the next input will be after $t_i + u_{i-1}$.

In order for an interface $\mathcal{I}$ to be plugable to the environment E, the contracts specified by $\mathcal{I}$ should fit the requirement of E at any time during the interaction. This means that t_{i+1} is the time to start a computation round $r_i = (\mathcal{V}_i, I_i)$ for $\mathcal{I}$ with a service specified by a timed design $\rho_i = p_i \vdash (R_i, c_i \leq \ell \leq d_i)$ such that $f_i \Rightarrow p_i$ (i.e. the input received by $\mathcal{I}$ satisfies the precondition of the

service), and $f_i \wedge R_i \Rightarrow g_i$ (the result from the service is expected by E), and $[t_{i+1} + c_i, t_{i+1} + d_i] \subseteq [t_{i+1} + l_i, t_{i+1} + u_i]$ (the computation time meets the time constraint of E). The interaction between environment E and interface $\mathcal{I}$ is depicted in Fig. 2.

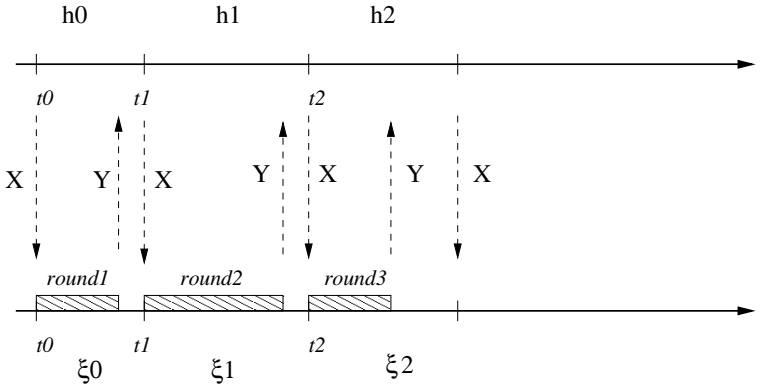


Fig. 2. Environment and interface interaction, contract ξi meets requirement hi at ti

Definition 6. *(Plugability) An interface $\mathcal{I} = (X', Y', \xi)$ is plugable to environment $E = (X, Y, h)$, denoted by $E\#\mathcal{I}$ iff $X' = X$, $Y' = Y$ and the following conditions are satisfied.*

1. *Let $h(\epsilon) = f_0 \vdash (g_0, l_0 \leq \ell \leq u_0)$, and $\xi(\epsilon) = \rho_0 = p_0 \vdash (R_0, c_0 \leq \ell \leq d_0)$. Then, $f_0 \Rightarrow p_0$, $f_0 \wedge R_0 \Rightarrow g_0$, and $t_0 = 0$, $[t_0 + c_0, t_0 + d_0] \subseteq [t_0 + l_0, t_0 + u_0]$. For any $\mathcal{V}_1$ such that $\mathcal{V}_1 \models f_0 \wedge R_0$ and interval $I_1 = [t_0, t_0']$ with $c_0 \leq length(I_1) \leq d_0$, for any $t_1 \geq t_0 + u_0$ the pair $(\mathcal{V}_1, t_1)$ is called reachable timed behavior of E w.r.t. $\mathcal{I}$, and $(\mathcal{V}_1, I_1)$ is called reachable state of $\mathcal{I}$ w.r.t. $(\mathcal{V}_1, t_1)$.*
2. *Let $w_n = (\mathcal{V}_1, t_1) \ldots (\mathcal{V}_n, t_n)$ be reachable timed behavior of E w.r.t. $\mathcal{I}$ such that $h(w_n)$ is defined, and $s_n = (\mathcal{V}_1, I_1) \ldots (\mathcal{V}_n, I_n)$ be reachable state of $\mathcal{I}$ w.r.t. w_n. Then, $\xi(s_n)$ is also defined. Furthermore, let $h(w_n) = f_n \vdash (g_n, l_n \leq \ell \leq u_n)$, and $\xi(s_n) = \rho_n = p_n \vdash (R_n, c_n \leq \ell \leq d_n)$. Then, $f_n \Rightarrow p_n$, $f_n \wedge R_n \Rightarrow g_n$, and $[t_n + c_n, t_n + d_n] \subseteq [t_n + l_{n-1}, t_n + u_{n-1}]$. For any $\mathcal{V}_{n+1}$ such that $\mathcal{V}_{n+1} \models f_n \wedge R_n$, for any interval $I_n = [t_n, t_n']$ with $c_n \leq length(I_n) \leq d_n$, and for any t_{n+1} such that $t_n + u_n \leq t_{n+1}$, $(\mathcal{V}_1, t_1) \ldots (\mathcal{V}_n, t_n)(\mathcal{V}_{n+1}, t_{n+1})$ is called reachable timed behavior of E w.r.t. $\mathcal{I}$, and $(\mathcal{V}_1, I_1) \ldots (\mathcal{V}_n, I_n)(\mathcal{V}_{n+1}, I_{n+1})$ is called reachable state of $\mathcal{I}$ w.r.t. $(\mathcal{V}_1, t_1) \ldots (\mathcal{V}_n, t_n)(\mathcal{V}_{n+1}, t_{n+1})$.*

Note that in this definition, for a pair $(\mathcal{V}_i, t_i)$ in a reachable behavior w.r.t. $\mathcal{I}$, t_{i-1} is the starting time of a round for which both input and output are given by value assignment $\mathcal{V}_i$. A necessary condition for the plugability is that for any reachable timed behavior w of E w.r.t. $\mathcal{I}$, and reachable state s of $\mathcal{I}$ w.r.t. w, if w is expandable, then s must not be a deadlock state of $\mathcal{I}$.

Given $E\#\mathcal{I}$, let $\mathcal{B}(E\#I)$ be the set of all reachable timed behaviors of E w.r.t. $\mathcal{I}$, and $\mathcal{S}(E\#I)$ be the set of all reachable states of $\mathcal{I}$ w.r.t. E. It is then $\mathcal{S}(E\#I) \subseteq \mathcal{R}(\mathcal{I})$ and $\mathcal{B}(E\#I) \subseteq \mathcal{B}(E)$. So, not all services from $\mathcal{I}$ are used by E, and the outputs from $\mathcal{I}$ may restrict the behaviors of E.

Lemma 1. *Let $\mathcal{I}$ be a real-time interface. There exists an environment $E_\mathcal{I}$ such that $E_\mathcal{I}\#\mathcal{I}$ and $\mathcal{S}(E_\mathcal{I}\#\mathcal{I}) = \mathcal{R}(\mathcal{I})$.*

Proof. Let $\mathcal{I} = (X, Y, \xi)$. For a reachable state $s_n \in \mathcal{R}(\mathcal{I})$ and $s_n = (\mathcal{V}_1, I_1)(\mathcal{V}_2, I_2)\dots(\mathcal{V}_n, I_n)$, we call the sequence $(\mathcal{V}_1, t_1)(\mathcal{V}_2, t_2)\dots(\mathcal{V}_n, t_n)$ a possible timed behavior w.r.t. s_n, where $t_0 = 0$, and for $i \geq 0$, t_{i+1} satisfy the following condition: let $\xi(\mathcal{V}_1, I_1)(\mathcal{V}_2, I_2)\dots(\mathcal{V}_{i-1}, I_{i-1}) = p_{i-1} \vdash (R_{i-1}, c_{i-1} \leq \ell \leq d_{i-1})$, then, t_{i+1} is any time point that satisfies $t_{i+1} \geq t_i + d_{i-1}$, where $\mathcal{V}_1, I_1)(\mathcal{V}_2, I_2)\dots(\mathcal{V}_{i-1}, I_{i-1}) = \epsilon$ when $i = 1$. The partial function h is defined as: if w is a possible behavior w.r.t. s, and $\xi(s) = \rho = p \vdash (R, c \leq \ell \leq d)$ then $h(w) = p \vdash (R, c \leq \ell \leq d)$. Let $E_\mathcal{I} = (X, Y, h)$. It is trivial to verify that $E_\mathcal{I}\#\mathcal{I}$ and $\mathcal{S}(E_\mathcal{I}\#I) = \mathcal{R}(\mathcal{I})$.

It is the plugability that helps to give a more natural definition for the refinement of interfaces. An interface $\mathcal{I}$ is said to be "better" than an interface $\mathcal{I}'$ iff $\mathcal{I}$ can replace $\mathcal{I}'$ in all cases of use of $\mathcal{I}'$.

Definition 7. *(Interface Refinement) An interface $\mathcal{I}$ is said to be a refinement of an interface $\mathcal{I}'$, denoted by $\mathcal{I} \sqsubseteq \mathcal{I}'$, iff for all environment E, if $\mathcal{I}'$ is plugable to E then $\mathcal{I}$ is also plugable to E.*

It turns out that this natural definition is equivalent to the classical one.

Theorem 1. *Let $\mathcal{I} = (X, Y, \xi)$ and $\mathcal{I}' = (X', Y', \xi')$ be real time interfaces. $\mathcal{I} \sqsubseteq \mathcal{I}'$ holds if and only if for all $s \in \mathcal{R}(\mathcal{I}') \cap \mathcal{R}(\mathcal{I})$ either $\xi(s) \sqsubseteq \xi'(s)$ holds or s is a deadlock state of $\mathcal{I}'$.*

Proof. The "only if" part of the theorem follows directly from Lemma 1. Since $E_{\mathcal{I}'}$ is also plugable to $\mathcal{I}$, if $\xi'(s)$ is defined, $\xi(s)$ must be defined and $\xi(s) \sqsubseteq \xi'(s)$ according to the definition of plugability.

The "if part" of the theorem is proved by induction on the length of common reachable states based on the definition of plugability. If $E = (X, Y, h)$ is plugable to $\mathcal{I}'$, the first item of Definition 6 is verified for $\mathcal{I}'$. Let $h(\epsilon) = f_0 \vdash (g_0, l_0 \leq \ell \leq u_0)$, and $\xi'(\epsilon) = \rho'_0 = p'_0 \vdash (R'_0, c'_0 \leq \ell \leq d'_0)$. Then, $f_0 \Rightarrow p'_0$, $f_0 \wedge R'_0 \Rightarrow g_0$, and $t_0 = 0$, $[t_0 + c'_0, t_0 + d'_0] \subseteq [t_0 + l_0, t_0 + u_0]$. Let $\xi(\epsilon) = \rho_0 = p_0 \vdash (R_0, c_0 \leq \ell \leq d_0)$. From the assumption $\xi(\epsilon) \sqsubseteq \xi'(\epsilon)$, it follows that $f_0 \Rightarrow p_0$, $f_0 \wedge R_0 \Rightarrow g_0$, $[t_0 + c_0, t_0 + d_0] \subseteq [t_0 + l_0, t_0 + u_0$. For any $\mathcal{V}_1$ such that $\mathcal{V}_1 \models f_0 \wedge R_0$ and interval $I_1 = [t_0, t'_0]$ with $c_0 \leq \mathit{length}(I_1) \leq d_0$ and for any $t_1 \geq t_0 + u_0$, $(\mathcal{V}_1, t_1)$ is a reachable timed behavior of E w.r.t. $\mathcal{I}$ and $\mathcal{I}'$, and $(\mathcal{V}_1, I_1)$ is a reachable state of both interfaces $\mathcal{I}$ and $\mathcal{I}'$ w.r.t. $(\mathcal{V}_1, t_1)$.

The second item of Definition 6 for E and $\mathcal{I}$ is verified exactly in the same way with the induction hypothesis.

It follows immediately from this theorem that:

Corollary 1. *Let $\mathcal{I}$ and $\mathcal{I}'$ be real time interfaces and $\mathcal{I} \sqsubseteq \mathcal{I}'$. Then, for any environment E such that $E\#\mathcal{I}'$ we have $\mathcal{R}(E\#\mathcal{I}) \subseteq \mathcal{R}(E\#\mathcal{I}')$*

Two interfaces $\mathcal{I}$ and $\mathcal{I}'$ are equivalent, denoted as $\mathcal{I} \equiv \mathcal{I}'$ iff $\mathcal{I} \sqsubseteq \mathcal{I}'$ and $\mathcal{I} \sqsubseteq \mathcal{I}'$.

Corollary 2. *Let $\mathcal{I} = (X, Y, \xi)$ and $\mathcal{I}' = (X', Y', \xi')$ be real time interfaces.*

1. *$\mathcal{I} \equiv \mathcal{I}'$ if and only if for any environment E, $E\#\mathcal{I}$ iff $E\#\mathcal{I}'$.*
2. *$\mathcal{I} \equiv \mathcal{I}'$ if and only if for all $s \in \mathcal{R}(\mathcal{I}) \cap \mathcal{R}(\mathcal{I}')$, $\xi(s) \equiv \xi'(s)$.*

3 Interface Composition

The most important operations on interfaces are composition operations. We consider two kinds of composition: parallel and sequential. Two interfaces $\mathcal{I}$ and $\mathcal{I}'$ can be put together either in parallel or in sequence to achieve a new interface that provide compound services. However, the execution time for the compound services need to considered carefully. From our intention, the time constraint in a timed design serves for the estimation of the execution time. The outputs from a component are supposed to be synchronous, and given at the end of the computation round. Therefore, the computation result coming out earlier must wait for those that have not been available yet. As soon as all the outputs are available, the computation round terminates. This consideration leads to the following definition of parallel and sequential composition of interfaces. Let us denote for a value assignment $\mathcal{V}$ and a set of variables V the restriction of $\mathcal{V}$ on V by $\mathcal{V}|_V$.

Definition 8. *(Parallel Composition)*
Let $\mathcal{I} = (X, Y, \xi)$ and $\mathcal{I}' = (X', Y', \xi')$ be two completely timeless interfaces such that $(X \cup Y) \cap (X' \cup Y') = \emptyset$. The parallel composition $\mathcal{I}||\mathcal{I}'$ is the interface $(X \cup X', Y \cup Y', \xi'')$ where $\xi'' : \mathcal{S}(X \cup X', Y \cup Y') \to \mathcal{D}(X \cup X', Y \cup Y')$ defined as follows. For $s = (\mathcal{V}_1, I_1'') \ldots (\mathcal{V}_n, I_n'') \in \mathcal{S}(X \cup X', Y \cup Y')$, $\xi''(s)$ is defined iff it is defined for all proper prefixes of s, and there exist time intervals $I_1, I_1', \ldots I_n, I_n'$ such that both

$$\xi((\mathcal{V}_1|_{X\cup Y}, I_1) \ldots (\mathcal{V}_n|_{X\cup Y}, I_n)) = p \vdash (R, c \leq \ell \leq d), \text{ and}$$
$$\xi'((\mathcal{V}_1|_{X'\cup Y'}, I_1') \ldots (\mathcal{V}_n|_{X'\cup Y'}, I_n')) = p' \vdash (R', c' \leq \ell \leq d')$$

are defined, and then

$$\xi''(s) = p \wedge p' \vdash (R \wedge R', \min\{c, c'\} \leq \ell \leq \max\{d, d'\})$$

and I_n'' satisfies that $\min\{c, c'\} \leq length(I_n'') \leq \max\{d, d'\}$ and I_n'' is after I_{n-1}''.

Note that the definition is well-formed (i.e. ξ'' is well defined) because $\mathcal{I}$ and $\mathcal{I}'$ are completely timeless. It also follows that $\mathcal{I}||\mathcal{I}'$ is also completely timeless.

By sequential composition of two interfaces, we mean that some inputs of the second are connected to some outputs of the first interface. To be defined,

an input can be connected to at most one output although several inputs may be connected to the same output. Given two interfaces $\mathcal{I} = (X, Y, \xi)$ and $\mathcal{I}' = (X', Y', \xi')$ such that $(X \cup Y) \cap (X' \cup Y') = \emptyset$. A connection from $\mathcal{I}$ to $\mathcal{I}'$ is a set of pairs $\theta \subseteq Y \times X'$ that satisfy $\forall (y, x), (y', x') \in \theta.(x = x' \Rightarrow y = y')$. Let $X_\theta = \{x \in X' | \exists y \in Y.(y, x) \in \theta\}$. A connection θ converses an assignment a over $((X \cup X') \setminus X_\theta) \cup Y \cup Y'$ to an assignment a^θ over $(X \cup X' \cup Y \cup Y')$ as: $a^\theta|_{((X \cup X') \setminus X_\theta) \cup Y \cup Y'} = a|_{((X \cup X') \setminus X_\theta) \cup Y \cup Y'}$, and for $x \in X_\theta$ we define $a^\theta(x) = a(y)$ where y is the unique element in Y such that $(y, x) \in \theta$. Let us denote $\nu_\theta = \bigwedge_{(y,x) \in \theta} x = y$.

Definition 9. *(Sequential Composition)*
Let $\mathcal{I} = (X, Y, \xi)$ and $\mathcal{I}' = (X', Y', \xi')$ be two completely timeless interfaces. Sequential composition of $\mathcal{I}$ and $\mathcal{I}'$ w.r.t connection θ, denoted by $\mathcal{I}.{}_\theta\mathcal{I}'$ is interface $\mathcal{I}'' = (X'', Y'', \xi'')$, where

- $X'' = (X \cup X') \setminus X_\theta$, $Y'' = Y \cup Y'$.
- *For $s = (\mathcal{V}_1, I_1'') \ldots (\mathcal{V}_n, I_n'') \in \mathcal{S}(X'', Y'')$, $\xi''(s)$ is defined iff it is defined for all proper prefix of s, and there exist time intervals $I_1, I_1', \ldots I_n, I_n'$ such that both*
$$\xi((a_1^\theta|_{X \cup Y}, I_1) \ldots (a_n^\theta|_{X \cup Y}, I_n)) = p \vdash (R, c \leq \ell \leq d), \text{and}$$
$$\xi'((a_1^\theta|_{X' \cup Y'}, I_1') \ldots (a_n^\theta|_{X' \cup Y'}, I_n')) = p' \vdash (R', c' \leq \ell \leq d')$$
are defined, and then
$$\xi''(s) = p \wedge \exists Y.(R \wedge p' \wedge \nu_\theta) \vdash (R \wedge R' \wedge \nu_\theta \wedge p', c + c' \leq \ell \leq d + d')$$
and I_n'' satisfies that $c + c' \leq length(I_n'') \leq d + d'$ and I_n'' is after I_{n-1}''.

In this definition, we assume that the second interface takes the outputs from the first as soon as they are available, and for those outputs that are not in use by the second as inputs the availability is delayed to be the same as the availability of the outputs from the second. Also, only the outputs from the first that satisfy the precondition for the second are produced as outputs. As for parallel composition, $\mathcal{I}.{}_\theta\mathcal{I}'$ is also a completely timeless interface.

Parallel composition and sequential composition of two interfaces $\mathcal{I}$ and $\mathcal{I}'$ are depicted in Fig. 3.

The following theorem follows immediately from Definitions 8 and 9.

Theorem 2. *Let $\mathcal{I}$, $\mathcal{I}'$ and $\mathcal{I}''$ be interfaces, θ and θ' be connections between $\mathcal{I}$ and $\mathcal{I}'$ and between $\mathcal{I}'$ and $\mathcal{I}''$ respectively. Then, the following hold: (1) $\mathcal{I}||\mathcal{I}' \equiv \mathcal{I}'||\mathcal{I}$, (2) $(\mathcal{I}||\mathcal{I}')||\mathcal{I}'' \equiv \mathcal{I}||(\mathcal{I}'||\mathcal{I}'')$, and (3) $(\mathcal{I}.{}_\theta\mathcal{I}').{}_{\theta'}\mathcal{I}'' \equiv \mathcal{I}.{}_\theta(\mathcal{I}'.{}_{\theta'}\mathcal{I}'')$*

4 Automata Interfaces

There are two issues for general real-time interfaces: how to represent interfaces finitely, and how to check if an interface is plugable to an environment.

In general, a real-time interface $\mathcal{I} = (X, Y, \xi)$ cannot be finitely representable since ξ is a function from an infinite set to an infinite set. When $\mathcal{I} = (X, Y, \xi)$

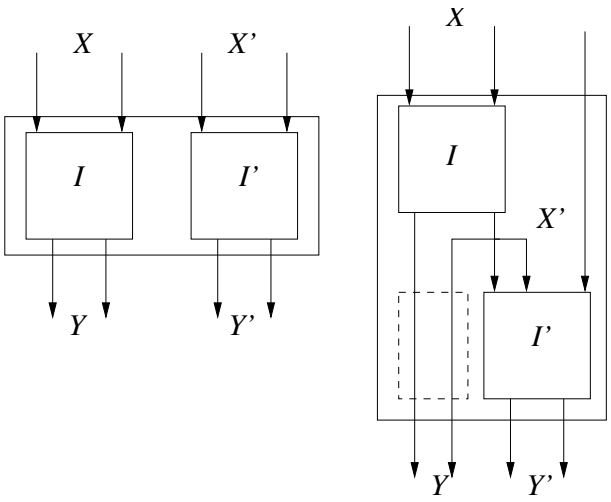


Fig. 3. Parallel and sequential composition

is finite, the range of ξ is finite. Therefore the coimage of ξ forms a finite partition $\pi = \{S_1, \ldots, S_k\}$ of $\mathbf{dom}(\xi)$. If we can decide if a state is a member of S_i then ξ is finitely representable. For the simplicity of presentation and in order to be practical, we will consider in this section only finite and completely timeless interfaces. With this restriction, automata seem to be among the best representations for finite interfaces.

Definition 10. *(Labeled Automata)*
A labeled automaton M is a tuple $M = (Q, X, Y, q_0, T, l_s, l_t)$, where Q is a finite set of locations, X and Y are sets of input and output variables respectively, $q_0 \in Q$ is an initial state of M, $T \subseteq Q \times Q$ is a set of transitions, and $l_s : Q \to \mathcal{D}(X, Y)$ and $l_t : T \to \mathcal{F}(X \cup Y)$ are labeling functions. l_s associates each location in M with a timed design, and l_t associates each transition in T with a guard formula. To make M deterministic, we assume that $l_t(q, q'') \wedge l_t(q, q') \Rightarrow false$ for any two different transitions (q, q') and (q, q'') with the same source state.

Let $V(X \cup Y)$ be the set of all valuations over the set of variables $X \cup Y$. A labeled automaton M can describe a partial function $g : V^*(X \cup Y) \to \mathcal{D}(X, Y)$ in the following way:

1. $g(\epsilon) = l_s(q_0)$, and ϵ is said to lead M to q_0.
2. For any sequence of valuation $s \in V^*(X \cup Y)$, if $g(s) = p \vdash (R, c \leq \ell \leq d)$, and s leads M to location q, for any computation round $(\mathcal{V}, I')$ and transition (q, q') such that $(\mathcal{V}, I') \models g(s)$ and $\mathcal{V} \models l_t(q, q')$, we have $g(s\mathcal{V}) = l_s(q')$ and $s\mathcal{V}$ is said to lead M to location q'.

The function described by a labeled automaton M is denoted by g_M. It is the definition of g_M that makes the $\mathbf{dom}(g_M)$ prefix closed.

Definition 11. *(Automata Interface)*

1. *Interface* $\mathcal{I} = (X, Y, \xi)$ *is said to be an automata interface iff there is a labeled automaton* M *such that for any state* $(\mathcal{V}_1, I_1) \dots (\mathcal{V}_k, I_k)$ *(*$k \geq 0$*, and the case* $k = 0$ *corresponds to the state* ϵ*), the* ξ *function value* $\xi((\mathcal{V}_1, I_1) \dots (\mathcal{V}_k, I_k))$ *is defined exactly when* $g_M(\mathcal{V}_i \dots \mathcal{V}_k)$ *is defined and* $\xi((\mathcal{V}_1, I_1) \dots (\mathcal{V}_k, I_k)) = g_M(\mathcal{V}_1 \dots \mathcal{V}_k)$ *provided that* $c_i \leq length(I_i) \leq d_i$*, where* $g_M(\mathcal{V}_1, \dots \mathcal{V}_i) = p_i \vdash (R_i, c_i \leq \ell \leq d_i)$*,* $i = 1, \dots, n$*. Such an automaton* M *is said to be a description of* $\mathcal{I}$*.*
2. *Environment* $E = (X, Y, h)$ *is said to be an automaton environment iff there is a label automaton* M *such that for any timed behavior* $(\mathcal{V}_1, t_1) \dots (\mathcal{V}_k, t_k)$*, the* h *function value* $h((\mathcal{V}_1, t_1) \dots (\mathcal{V}_k, t_k))$ *is defined exactly when* $g_M(\mathcal{V}_i \dots \mathcal{V}_k)$ *is defined and* $h((\mathcal{V}_1, t_1) \dots (\mathcal{V}_k, t_k)) = g_M(\mathcal{V}_i \dots \mathcal{V}_k)$ *provided that* $t_{i+1} \geq t_i + d_i$*,* $t_1 = 0$*, where* $g_M(\mathcal{V}_1, \dots \mathcal{V}_i) = p_i \vdash (R_i, c_i \leq \ell \leq d_i)$*,* $i = 1, \dots, n$*. Such an automaton* M *is said to be a description of* E*.*

Example 2. Let us consider a GPS as a real-time interface $\mathcal{I}$, and its user as an environment E. The interface has the input variable set $\{usage, destination\}$, and the output variable set $\{display\}$. The automata representation of $\mathcal{I}$ and E are depicted in Fig 4. The label of states of the interface $\mathcal{I}$ automaton are: $l_s(s1) = true \vdash (display = idle, 0 \leq \ell \leq 1)$, $l_s(s2) = true \vdash (display = current\_position, 0 \leq \ell \leq 1)$, and $l_s(s3) = legal(destination) \vdash (display = route\_to\_destination, 1 \leq \ell \leq 2)$. The label of states of the environment E automaton are: $l_s(r1) = (usage = 1) \vdash (display = idle, 0 \leq \ell \leq 1)$, $l_s(r2) = (usage = 1) \vdash (display = current\_position,\ 0 \leq \ell \leq 1)$, and $l_s(s3) = (legal(destination) \wedge\ usage = 0 \vdash (display = route\_to\_destination, 0 \leq \ell \leq 3)$. The user gives the input "$usage = 1$" to the interface to turn on the system, then again gives the input "$usage = 1$" to the interface to get "*current_position*", and then gives the input "$usage = 1$" and a "*destination*" to the interface to get a "*route_to_destination*" to the destination, and finally an input "$usage = 0$" to turn off the system.

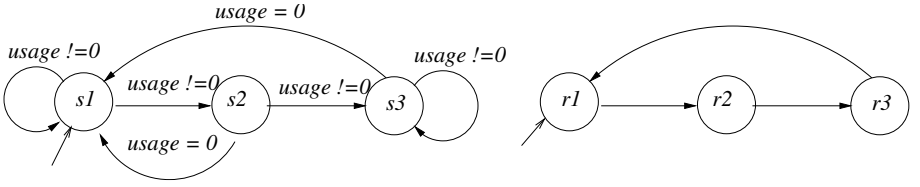


Fig. 4. An automata interface and an automata environment

In the interface theory, it is important that the plugability of an interface to an environment is decidable. This is possible for automata interfaces and environments.

Theorem 3. *Let $\mathcal{I} = (X, Y, \xi)$ be an automaton interface described by automaton $M = (Q, X, Y, q_0, T, l_s, l_t)$ and $E = (X, Y, h)$ be an automata environment described by automaton $M' = (Q', X, Y, q'_0, T', l'_s, l'_t)$. $E\#\mathcal{I}$ if and only if there is a correspondence f from Q' to Q (one may correspond to many) that satisfies:*

1. *$f(q'_0) = \{q_0\}$*
2. *For any $q \in f(q')$, $l_s(q) \sqsubseteq l'_s(q')$*
3. *Let $q \in f(q')$ and $l_s(q) = p \vdash (R, c \le \ell \le d)$ and $l'_s(q) = p' \vdash (R', c' \le \ell \le d')$. For any $r' \in Q'$ such that $F_{(q',r')} \widehat{=} (p' \wedge R \wedge l'_t(q', r'))$ is satisfiable, let $\phi(q) = \{r \mid (q, r) \in T$ and $l_t(q, r) \wedge F_{(q',r')}$ is satisfiable$\}$. Then $F_{(q',r')} \Rightarrow \bigvee_{r \in \phi(q)} l_t(q, r)$ and $\phi(q) \subseteq f(r')$.*

Proof. The "if part" is proved by induction on reachable behavior of E and direct check of the definition of plugability. The "only if" part follows from an inductive construction of a correspondence f from the inductive definition of plugability between E and $\mathcal{I}$. We omit the proof details here.

In Example 2, the correspondence f defined as $f(r_i) = s_i$, $i = 1, 2, 3$. f satisfies the conditions in Theorem 3. Therefore, $\mathcal{I}\#E$.

As a special case of Theorem 3, let $Q' \subseteq Q$, and let $\pi \in X$ be a special distinct input variable, π does not occur in any timed design in the labels of automata M and M', and $\mathbf{dom}(\pi) = Q'$. Let $l_t(q', q) = l'_t(q', q) \widehat{=} (\pi = q)$, $l_s(q) = l'_s(q)$ for all $q', q \in Q'$, let f be the identifying mapping on Q'. Then, by Theorem 3, if the set of all transition sequences (paths) starting from initial state of M' is a subset of the set of all transition sequences starting from initial state of M, $E\#\mathcal{I}$ holds. In this case, the set of all transition sequences starting from initial state of M play the rôle of interaction protocols. If the behavior of E represented by M' "obeys" these protocols, E is plugable to $\mathcal{I}$.

It is easy to design an algorithm for checking the conditions of Theorem 3 to decide the plugability of a given automata interface to a given automata environment E.

Algorithm. Checking plugability

Input: Automata interface $\mathcal{I}$ described by automaton $M = (Q, X, Y, q_0, T, l_s, l_t)$, and an automata environment $E = (X, Y, h)$ described by automaton $M' = (Q', X, Y, q'_0, T', l'_s, l'_t)$.

Output: "Yes" if $E\#\mathcal{I}$, and "no" otherwise.

Method: Let $f \subseteq Q' \times Q$. f is initialized as $f = \{(q'_0, q_0)\}$ and (q'_0, q_0) is unmarked. Carry out following steps.

1. If no unmarked element is found in f, stop with the output "yes". Otherwise, take an unmarked element $(q', q) \in f$ and mark it.
2. If $l_s(q) \not\sqsubseteq l'_s(q')$, stop with the output "no". Otherwise, let $l_s(q) = p \vdash (R, c \le \ell \le d)$ and $l'_s(q') = p' \vdash (R', c' \le \ell \le d')$. For any $r' \in Q'$ such that $F_{(q',r')} \widehat{=} p' \wedge R \wedge l'_t(q', r')$ is satisfiable, let $\phi(q) \widehat{=} \{r \mid (q, r) \in T$ and $l_t(q, r) \wedge F_{(q',r')}$ is satisfiable$\}$. If $F_{(q',r')} \Rightarrow \bigvee_{r \in \phi(q)} l_t(q, r)$ is false, stop with the output "no". Otherwise, add unmarked pairs (r', r) to f for any $r \in \phi(q)$.
3. Goto Step 1.

In this algorithm we have to check the satisfiability of some logic formulas. When the domain of the variables in the formula are finite, satisfiability can be checked with SAT solvers.

As a corollary of Theorem 1, we have:

Theorem 4. *Let $\mathcal{I} = (X, Y, \xi)$ be an automata interface described by automaton $M = (Q, X, Y, q_0, T, l_s, l_t)$ and $\mathcal{I}' = (X, Y, \xi')$ be an automata interface described by automaton $M' = (Q', X, Y, q'_0, T', l'_s, l'_t)$. $\mathcal{I} \sqsubseteq \mathcal{I}'$ if and only if there is a correspondence f from Q' to Q (one may correspond to many) that satisfies:*

1. *$f(q'_0) = \{q_0\}$*
2. *For any $q \in f(q')$, $l_s(q) \sqsubseteq l'_s(q')$*
3. *Let $q \in f(q')$ and $l_s(q) = p \vdash (R, c \leq \ell \leq d)$ and $l'_s(q) = p' \vdash (R', c' \leq \ell \leq d')$. For any $r' \in Q'$ such that $F_{(q',r')} = p' \wedge R \wedge l'_t(q', r')$ is satisfiable, let $\phi(q) = \{r \mid (q, r) \in T$ and $l_t(q, r) \wedge F_{(q'r')}$ is satisfiable$\}$. Then $F_{(q',r')} \Rightarrow \bigvee_{r \in \phi(q)} l_t(q, r)$ and $\phi(q) \subseteq f(r')$.*

Proof. It follows from Theorem 1 that $\mathcal{I} \sqsubseteq \mathcal{I}'$ if and only if the environment $E_{\mathcal{I}'}$ is plugable to $\mathcal{I}$. Hence, Theorem 4 follows immediately Theorem 3.

5 Discussion and Related Work

As it is said in the introduction of this paper, the model presented in this paper is an extension of the model presented in [7] for real-time interfaces. The main difference is, here we use UTP timed designs to specify services, and services are defined only on reachable states. In addition to time extension, we found that using UTP is much more convenient than using other notations. We focus in this paper on checking the refinement and plugability and finite representations instead of providing a theory of real-time interfaces.

In the literature, there have been several studies on real-time interfaces such as [9–12]. In those papers, the authors have a focus on the analysis of schedulability from the task arrival rates. Our focus in this paper is different. We aim to provide a formal model that can help to do the shedulability analysis. Compared to our previous work [13, 3, 5], the model in this paper can capture the relation between interaction protocols and services. However, in this paper we ignored the relationship between resources and worst case execution times for the simplicity. This model can be easily extended to capture that relation. In our previous work [14], we checked at runtime if an environment follows the interaction protocol specified by a component. In this paper, we provide an algorithm to check at compile time the conformance between an environment and a component in term of both interaction protocols and their related services. Though the complexity of our checking algorithm is a bit high, we believe that it can be implemented with the help of advanced SAT solvers.

6 Conclusion

We have presented a real-time interface model using UTP. The services of an interface are specified as a timed design in UTP, and depend on the current states of the interface. The model is general enough to capture all aspects of interface specification such as functionality, non-functionality, interaction protocol and the relation between them. We have also considered some operations on real-time interfaces. The operations are used in the incremental development of component-based embedded systems. We have also considered a finite representation of interfaces by labeled automata. We showed that the syntactical definition of interface refinement is consistent to the semantic one. We have shown that when interfaces and environments can be represented by automata, checking the plugability between interfaces and environments can be done effectively using SAT solvers. We believe that for finite interfaces that are not completely timeless, labeled time automata will be a good candidate for representing them, and similar results for checking the plugability can be obtained.

What we have left out in this paper is the analysis of the timed behaviors of environments when a component is plugged to, and to carry out verification of some real-time properties and the schedulability when there are several threads running in parallel in environments. This will be our future consideration.

References

1. de Alfaro, L., Henzinger, T.A.: Interface Automata. In: ACM Symposium on Foundation of Software Engineering, FSE (2001)
2. Doyen, L., Henzinger, T.A., Jobstmann, B., Petrov, T.: Interface theories with component reuse. In: EMSOFT, pp. 79–88 (2008)
3. Van Hung, D.: Toward a formal model for component interfaces for real-time systems. In: Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems, FMICS 2005, pp. 106–114. ACM, New York (2005)
4. 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
5. Ledang, H., Van Hung, D.: Timing and concurrency specification in component-based real-time embedded systems development. In: TASE, pp. 293–304. IEEE Computer Society (2007)
6. 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)
7. Tripakis, S., Lickly, B., Henzinger, T.A., Lee, E.A.: A theory of synchronous relational interfaces. ACM Transactions on Programming Languages and Systems 33(4) (2011)
8. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall Series in Computer Science, Prentice Hall (1998)
9. Wang, S., Rho, S., Mai, Z., Bettati, R., Zhao, W.: Real-time component-based systems. In: Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium, RTAS 2005, pp. 1080–1812. IEEE Computer Society (2005)

10. Henzinger, T.A., Matic, S.: An interface algebra for real-time components. In: IEEE Real Time Technology and Applications Symposium, pp. 253–266 (2006)
11. Delahaye, B., Fahrenberg, U., Henzinger, T.A., Legay, A., Ničković, D.: Synchronous interface theories and time triggered scheduling. In: Giese, H., Rosu, G. (eds.) FMOODS/FORTE 2012. LNCS, vol. 7273, pp. 203–218. Springer, Heidelberg (2012)
12. Wiklander, J., Eliasson, J., Kruglyak, A., Lindgren, P., Nordlander, J.: Enabling component-based design for embedded real-time software. Journal of Computers 4(12), 1039–1321 (2009)
13. Van Hung, D., Vu Anh, B.: Model checking real-time component based systems with blackbox testing. In: RTCSA, pp. 76–79 (2005)
14. Truong, A.-H., Trinh, T.-B., Van Hung, D., Nguyen, V.-H., Trang, N.T.T., Hung, P.D.: Checking interface interaction protocols using aspect-oriented programming. In: Software Engineering and Formal Methods (SEFM 2008), pp. 382–386 (2008)

Some Fixed-Point Issues in PPTL*

Zhenhua Duan**, Qian Ma, Cong Tian, and Nan Zhang

Institute of Computing Theory and Technology and ISN Laboratory
P.O. Box 177, Xidian University, Xi'an 710071, P.R. China
{zhhduan,qma,ctian,nzhang}@mail.xidian.edu.cn

Abstract. In Propositional Projection Temporal Logic (PPTL), a *well-formed* formula is generally formed by applying rules of its syntax finitely many times. However, under some circumstances, although formulas such as ones expressed by *index set expressions*, are constructed via applying rules of the syntax infinitely many times, they are possibly still well-formed. With this motivation, this paper investigates the relationship between formulas specified by index set expressions and concise syntax expressions by means of fixed-point induction approach. Firstly, we present two kinds of formulas, namely $\bigvee_{i\in N_0} \bigcirc^i P$ and $\bigvee_{i\in N_0} P^i$, and prove they are indeed well-formed by demonstrating their equivalence to formulas $\Diamond P$ and P^+ respectively. Further, we generalize $\bigvee_{i\in N_0} \bigcirc^i Q$ to $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q$ and explore solutions of an abstract equation $X \equiv Q \vee P \wedge \bigcirc X$. Moreover, we equivalently represent 'U' (strong until) and 'W' (weak until) constructs in Propositional Linear Temporal Logic within PPTL using the index set expression techniques.

1 Introduction

Temporal Logic (TL) [11] is a useful formalism for specifying properties of concurrent systems. Variants of TL have been proposed, such as Linear Temporal logic (LTL) [13], Computational Tree Logic (CTL) [1], Interval Temporal Logic (ITL) [12], and Projection Temporal Logic (PTL) [3,4] etc. Propositional PTL (PPTL) [3] is a propositional subset of PTL with a usual next construct $\bigcirc P$ and a new projection construct $(P_1, \ldots, P_m)$ prj Q as its basic constructs. At present, a decision procedure [6] and an axiomatic system [7] for PPTL are available, which enables PPTL to be utilized in both model checking [2] and theorem proving [9,8].

In general, a *well-formed* formula in PPTL is obtained through applying rules of its syntax finitely many times. However, under some circumstances, although formulas such as $\bigvee_{i\in N_0} \bigcirc^i P$ with N_0 the set of non-negative integers (called *index set expression*), are formed via applying rules of the syntax countably infinitely many times, they are actually well-formed since their equivalent well-formed PPTL formulas can be found. Thus, we are motivated to identify some such formulas and prove they are indeed well-formed.

* This research is supported by the NSFC Grant Nos. 61133001, 61272118, 61272117, 61202038, 91218301 and National Program on Key Basic Research Project (973 Program) Grant No. 2010CB328102.

** Corresponding author.

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 151–165, 2013.

Our contributions are three-fold: (1) We present two kinds of formulas with index set expressions, namely $\bigvee_{i\in N_0} \bigcirc^i P$ and $\bigvee_{i\in N_0} P^i$, and prove they are indeed well-formed by means of demonstrating their equivalence to $\Diamond P$ and P^+ respectively with fixed-point induction method [15]. (2) We generalize $\bigvee_{i\in N_0} \bigcirc^i Q$ to $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q$ and explore the least and great fixed-points of the abstract equation $X \equiv Q \vee P \wedge \bigcirc X$. (3) We equivalently represent the operator 'U' (strong until) and 'W' (weak until) of Propositional LTL (PLTL) within PPTL using the index set expression technique.

This paper is organized as follows. Section 2 briefly introduces PPTL. In Section 3, some fixed-point issues concerning $\bigvee_{i\in N_0} \bigcirc^i P$, $\bigvee_{i\in N_0} P^i$ and $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q$ are given. Moreover, Section 4 is devoted to equivalently denoting 'U' and 'W' constructs of PLTL within PPTL. Finally, conclusions are drawn in Section 5.

2 Propositional Projection Temporal Logic

Propositional Projection Temporal Logic (PPTL) [3,4] is an extension of Propositional ITL (PITL) [12] with a new projection construct [5]. Let *Prop* be a countable set of atomic propositions and $B = \{true, false\}$ the boolean domain. Usually, we use small letters, possibly with subscripts, like *p,q,r* to denote atomic propositions and capital letters, possibly with subscripts, like *P, Q, R* to represent general PPTL formulas. Then the formulas of PPTL are defined by the following grammar:

$$P ::= p \mid \neg P \mid P_1 \wedge P_2 \mid \bigcirc P \mid (P_1, \ldots, P_m)\ \mathsf{prj}\ P \mid P^+$$

where $p \in Prop$, $\bigcirc$ (next), $+$ (chop-plus) and prj (projection) are temporal operators, and $\neg, \wedge$ are similar as that in classical propositional logic.

We define a *state* s over *Prop* to be a mapping from *Prop* to B, $s : Prop \to B$. We write $s[p]$ to denote the valuation of p at state s. An *interval* $\sigma = \langle s_0, s_1, ...\rangle$ is a non-empty sequence of states, which can be finite or infinite. The length of σ, $|\sigma|$, is the number of states in σ minus one if σ is finite; otherwise it is ω. To have a uniform notation for both finite and infinite intervals, we will use *extended integers* as indices, that is, $N_\omega = N_0 \cup \{\omega\}$ and extend the comparison operators, $=, <, \leq$, to N_ω by considering $\omega = \omega$ and for all $i \in N_0, i < \omega$. Moreover, we write $\preceq$ as $\leq -\{(\omega, \omega)\}$. Let $\sigma = \langle s_0, s_1, \ldots\rangle$ be an interval and $r_1, \ldots, r_h$ be integers ($h \geq 1$) such that $0 \leq r_1 \leq \ldots \leq r_h \preceq |\sigma|$. The projection of σ onto $r_1, \ldots, r_h$ is the *projected interval*, $\sigma \downarrow (r_1, \ldots, r_h) \stackrel{\text{def}}{=} \langle s_{t_1}, s_{t_2}, \ldots, s_{t_l}\rangle$, where $t_1, \ldots, t_l$ are attained from $r_1, \ldots, r_h$ by deleting all duplicates. In other words, $t_1, \ldots, t_l$ is the longest strictly increasing subsequence of $r_1, \ldots, r_h$. The concatenation($\cdot$) of an interval σ with another interval σ' is represented by $\sigma \cdot \sigma'$ (not sharing any states).

An *interpretation* is a tuple $\mathcal{I} = (\sigma, k, j)$, where $\sigma = \langle s_0, s_1, \ldots\rangle$ is an interval, k is a non-negative integer, and j is an integer or ω, such that $0 \leq k \preceq j \leq |\sigma|$. We write (σ, k, j) to mean that a formula is interpreted over a subinterval $\sigma_{k,\ldots,j}$ with the current state being s_k. We utilize I^k_{prop} to stand for the state interpretation at state s_k. The satisfaction relation $\models$ for formulas is given as follows:

$$\mathcal{I} \models p \text{ iff } s_k[p] = I^k_{prop}[p] = true$$
$$\mathcal{I} \models \neg P \text{ iff } \mathcal{I} \not\models P$$
$$\mathcal{I} \models P_1 \wedge P_2 \text{ iff } \mathcal{I} \models P_1 \text{ and } \mathcal{I} \models P_2$$

$\mathcal{I} \models \bigcirc P$ iff $k < j$ and $(\sigma, k+1, j) \models P$
$\mathcal{I} \models (P_1, \ldots, P_m)$ prj P iff there exist integers $r_0, \ldots, r_m$, and $k = r_0 \leq \ldots \leq r_{m-1} \preceq r_m \leq j$ such that $(\sigma, r_{l-1}, r_l) \models P_l$ for all $1 \leq l \leq m$ and $(\sigma', 0, |\sigma'|) \models P$ for σ' given by :
(1) $r_m < j$ and $\sigma' = \sigma \downarrow (r_0, \ldots, r_m) \cdot \sigma_{(r_m+1,\ldots,j)}$
(2) $r_m = j$ and $\sigma' = \sigma \downarrow (r_0, \ldots, r_h)$ for some $0 \leq h \leq m$
$\mathcal{I} \models P^+$ iff there are finitely many integers $r_0, \ldots, r_n$ and $k = r_0 \leq r_1 \leq \ldots \leq r_{n-1} \preceq r_n = j$ $(n \geq 1)$ such that $(\sigma, r_{l-1}, r_l) \models P$ for all $1 \leq l \leq n$; or $j = \omega$ and there are infinitely many integers $k = r_0 \leq r_1 \leq r_2 \leq \ldots$ such that $\lim_{i \to \infty} r_i = \omega$ and $(\sigma, r_{l-1}, r_l) \models P$ and for all $l \geq 1$.

A formula *P* is satisfied by an interval σ, signified by $\sigma \models P$ if $(\sigma, 0, |\sigma|) \models P$. A formula *P* is called *satisfiable* if $\sigma \models P$ for some σ. Furthermore, *P* is said to be *valid*, denoted by $\models P$, if $\sigma \models P$ for all intervals σ.

Some derived formulas of PPTL are shown below, which are explained in [4,3]. The abbreviations true, false, $\vee$, $\rightarrow$ and $\leftrightarrow$ are defined as usual.

ε	$\stackrel{\text{def}}{=} \neg \bigcirc \mathsf{true}$	P^*	$\stackrel{\text{def}}{=} P^+ \vee \varepsilon$
$\Diamond P$	$\stackrel{\text{def}}{=} (\mathsf{true}, P)\,\mathsf{prj}\ \varepsilon$	more	$\stackrel{\text{def}}{=} \neg \varepsilon$
$\Box P$	$\stackrel{\text{def}}{=} \neg \Diamond \neg P$	$\mathsf{fin}(P)$	$\stackrel{\text{def}}{=} \Box(\varepsilon \rightarrow P)$
$\mathsf{halt}(P)$	$\stackrel{\text{def}}{=} \Box(\varepsilon \leftrightarrow P)$	$\mathsf{keep}(P)$	$\stackrel{\text{def}}{=} \Box(\neg\varepsilon \rightarrow P)$
$P\,;Q$	$\stackrel{\text{def}}{=} (P, Q)\,\mathsf{prj}\ \varepsilon$	$P\,;_{\mathsf{w}} Q$	$\stackrel{\text{def}}{=} (P\,;Q) \vee (P \wedge \Box\mathsf{more})$
fin	$\stackrel{\text{def}}{=} \Diamond\varepsilon$	$\mathsf{len}(n)$	$\stackrel{\text{def}}{=} \begin{cases} \varepsilon & \text{if } n = 0 \\ \bigcirc\mathsf{len}(n-1) & \text{if } n > 1 \end{cases}$
inf	$\stackrel{\text{def}}{=} \Box\mathsf{more}$	$P \parallel Q$	$\stackrel{\text{def}}{=} (P \wedge (Q\,;\mathsf{true})) \vee (Q \wedge (P\,;\mathsf{true}))$

Commonly, $\models \Box(P \leftrightarrow Q)$ is represented by $P \equiv Q$ (*strong equivalence*), meaning that *P* and *Q* have the same truth values at all states in every model.

3 Fixed-Point Issues

A *well-formed* formula in PPTL is generally constructed by applying rules of the syntax finitely many times. However, although some formulas are formed via applying rules of the syntax countably infinitely many times, such as *index set expressions* (e.g. $\bigvee_{i \in N_0} \bigcirc^i P$), they are still well-formed due to the existence of their equivalent well-formed formulas. In this section, we identify two types of such formulas and prove they are indeed well-formed by means of the fixed-point induction approach [15]. Besides, we generalize one of them to a more generic form and investigate some related properties of an abstract equation $X \equiv Q \vee P \wedge \bigcirc X$.

3.1 Two Kinds of Index Set Expressions

(1) $\bigvee_{i \in N_0} \bigcirc^i P$

On one hand, $\bigvee_{i \in N_0} \bigcirc^i P \equiv P \vee \bigcirc P \vee \bigcirc^2 P \vee \bigcirc^3 P \vee \ldots$, is a disjunction of countably infinitely many $\bigcirc^i P$, where $\bigcirc^0 P \equiv P$. Intuitively, this formula means P

necessarily holds at some state from now on over an interval, which might be specified by the operator $\Diamond$. On the other hand, $\Diamond P$ indeed can be rewritten as:

$$\begin{aligned}\Diamond P &\equiv P \vee \bigcirc \Diamond P \\ &\equiv P \vee \bigcirc (P \vee \bigcirc \Diamond P) \\ &\equiv P \vee \bigcirc P \vee \bigcirc^2 \Diamond P \\ &\quad \ldots \\ &\equiv P \vee \bigcirc P \vee \bigcirc^2 P \vee \bigcirc^3 P \vee \ldots \qquad (\star)\end{aligned}$$

From the above, we observe that $\bigvee_{i \in N_0} \bigcirc^i P$ seems to be equivalent to $\Diamond P$, which will be affirmed in Theorem 1.

(2) $\bigvee_{i \in N_0} P^i$

For a chop formula $P_1 \,;\, \ldots \,;\, P_m$, if all $P_i \equiv P$ $(1 \leq i \leq m)$, we can acquire:

$$\underbrace{P \,;\, \ldots \,;\, P}_{m \ \ times}$$

which is briefly represented as P^m. For instance, $P^1 \equiv P$, $P^2 \equiv P \,;\, P$, and particularly $P^0 \equiv \mathsf{false}$. Thus, $\bigvee_{i \in N_0} P^i$ denotes $P \vee (P \,;\, P) \vee (P \,;\, P \,;\, P) \vee \ldots$. Further, we have the equation about P^+:

$$\begin{aligned}P^+ &\equiv P \vee (P \,;\, P^+) \\ &\equiv P \vee (P \,;\, (P \vee (P \,;\, P^+))) \\ &\equiv P \vee P \,;\, P \vee P \,;\, (P \,;\, P^+)) \\ &\quad \ldots \\ &\equiv P \vee (P \,;\, P) \vee (P \,;\, P \,;\, P) \vee \ldots\end{aligned}$$

Hence, we can declare that $\bigvee_{i \in N_0} P^i \equiv P^+$ in Theorem 1.

Theorem 1. *The following logical laws hold:*

1. $\bigvee_{i \in N_0} \bigcirc^i P \equiv \Diamond P$
2. $\bigvee_{i \in N_0} P^i \equiv P^+$

Proof. The two laws can be proved in an analogous way and we only prove $\bigvee_{i \in N_0} \bigcirc^i P \equiv \Diamond P$. The proof proceeds by fixed-point induction approach.

We firstly define $D = \{d_{-1}, d_0, \ldots, d_n, \ldots, d_\omega\}$, where $d_{-1} = \bigcirc^{-1} P = \mathsf{false}$, $d_i = \bigcirc^0 P \vee \ldots \vee \bigcirc^i P$ $(i \in N_0)$, $d_\omega = \bigvee_{i \in N_0} \bigcirc^i P$. Let $N_\omega = N_0 \cup \{\omega\}$ with $\omega = \omega, \omega + c = \omega$ (c is an integer) and for all $i \in N_0, i < \omega$. Further, a binary relation $\preccurlyeq$ over D is formalized as

$$d_i \preccurlyeq d_j \text{ iff } i \leq j \ (i, j \in N_\omega \cup \{-1\})$$

Moreover, let $f : D \to D$ be a function given by

$$f(d_i) = P \vee \bigcirc d_i$$

Then $f(d_i) = P \vee \bigcirc(P \vee \ldots \vee \bigcirc^i P) = d_{i+1}$ for $i \in \{-1\} \cup N_0$, and $f(d_\omega) = P \vee \bigcirc(\bigvee_{i \in N_0} \bigcirc^i P) = P \vee \bigvee_{i \in N_0} \bigcirc^{i+1} P = \bigvee_{i \in N_0} \bigcirc^i P = d_\omega$. Obviously, we have $d_i \sqcup d_j = d_i \vee d_j = d_j$ if $d_i \preccurlyeq d_j$.

1. $\bigvee_{i \in N_0} \bigcirc^i P$ is the least fixed-point of f

(1) $(D, \preccurlyeq)$ is a complete partial order

$(D, \preccurlyeq)$ is a partial order, since it satisfies the properties below:

- reflexivity: for all $d_i \in D$, clearly we have $d_i \preccurlyeq d_i$ due to $i \leq i$.
- anti-symmetry: if $d_i \preccurlyeq d_j$ and $d_j \preccurlyeq d_i$, then we obtain $i \leq j$ and $j \leq i$, leading to $i = j$. Hence $d_i = d_j$.
- transitivity: if $d_i \preccurlyeq d_j$, $d_j \preccurlyeq d_k$, then $i \leq j \leq k$. Thus, $d_i \preccurlyeq d_k$.

Furthermore, for any non-empty subset $S = \{d_{i_1}, \ldots, d_{i_n}\}$ of D, where $-1 \leq i_1 \leq \ldots \leq i_{n-1} \preceq i_n \leq \omega$, if S is finite, as $d_i \sqcup d_j = d_j$, we obtain the least upper bound $d_{i_n} \in D$, where i_n is the biggest index in S; otherwise, S is infinite, there exists a least upper bound $\bigsqcup_{i_n \in N_0} d_{i_n} = \bigvee_{i_n \in N_0} d_{i_n} = (P \vee \bigcirc P \vee \ldots \vee \bigcirc^{i_1} P) \vee (P \vee \bigcirc P \vee \ldots \vee \bigcirc^{i_2} P) \vee \ldots = d_\omega$, which obviously belongs to D. Thus, $(D, \preccurlyeq)$ is a complete partial order.

(2) f is a continuous function

Suppose that $d_i \preccurlyeq d_j$, then $i \leq j$, so $i + 1 \leq j + 1$. As a result,

$$f(d_i) = d_{i+1} \preccurlyeq d_{j+1} = f(d_j)$$

Hence, f is monotonic. Moreover, for an arbitrary ω-chain in D, $d_{i_1} \preccurlyeq d_{i_2} \preccurlyeq \ldots \preccurlyeq d_{i_n} \preccurlyeq \ldots$, if there exists an element d_{i_n} such that $d_{i_n} \preccurlyeq d_{i_n} \preccurlyeq \ldots$, it is apparent that d_{i_n} is the least upper bound of this ω-chain. Thus, we have

$$\begin{aligned} f(\bigsqcup_{i_n \in N_0} d_{i_n}) = f(d_{i_n}) = d_{i_n+1} = d_{i_n} \sqcup d_{i_n+1} &= \bigsqcup_{i_n \in N_0} d_{i_n} \sqcup d_{i_n+1} \\ &= d_0 \sqcup \bigsqcup_{i_n \in N_0} d_{i_n+1} \\ &= \bigsqcup_{i_n \in N_0} d_{i_n+1} = \bigsqcup_{i_n \in N_0} f(d_{i_n}) \end{aligned}$$

Otherwise, we can obtain the following:

$$\begin{aligned} \bigsqcup_{i_n \in N_0} f(d_{i_n}) = \bigsqcup_{i_n \in N_0} d_{i_n+1} = d_0 \sqcup \bigsqcup_{i_n \in N_0} d_{i_n+1} &= \bigsqcup_{i_n \in N_0} d_{i_n} \\ = d_\omega = f(d_\omega) = f(\bigsqcup_{i_n \in N_0} d_{i_n}) & \end{aligned}$$

Therefore, f is a continuous function. Hence, by Kleene Fixed-point Theorem [15,10], there exists a least fixed-point:

$$\begin{aligned} fix_\mu(f) = \bigsqcup_{n \in N_0} f^n(d_{-1}) = \bigsqcup_{n \in N_0} d_{n-1} = d_{-1} \sqcup \bigsqcup_{n \in N_0} d_n & \\ = \bigsqcup_{n \in N_0} d_n = d_\omega = \bigvee_{i \in N_0} \bigcirc^i P & \end{aligned}$$

2. $\bigvee_{i \in N_0} \bigcirc^i P$ is equivalent to $\Diamond P$

By the equation ($\star$), each $P \vee \bigcirc P \vee \ldots \vee \bigcirc^i P$ ($i \in N_0$) is called a *prefix* of $\Diamond P$. Particularly, false is also a prefix of $\Diamond P$. Then we construct a subset B of D as follows:

$$B = \{d_i |\ d_i \in D \text{ and } d_i \text{ is a prefix of the formula } \Diamond P\}$$

For any ω-chain $d_{i_1} \preccurlyeq d_{i_2} \preccurlyeq \ldots \preccurlyeq d_{i_n} \preccurlyeq \ldots$ in D, suppose each $d_{i_n} = P \vee \bigcirc P \vee \ldots \vee \bigcirc^{i_n} P$ ($i_n \in N_0$) is a prefix of $\Diamond P$, i.e. $d_{i_n} \in B$. Then as $d_i \sqcup d_{i+1} = d_{i+1}$, $\bigsqcup_{i_n \in N_0} d_{i_n} = P \vee \bigcirc P \vee \ldots \vee \bigcirc^i P \vee \ldots$ is also a prefix of $\Diamond P$ and belongs to B. Thus, we can obtain that B is an inclusive subset of D.

Moreover, the bottom element false is obviously a prefix of $\Diamond P$, thus false $\in B$. With the assumption of $d_i \in B$, when $i \in N_0 \cup \{-1\}$, since $f(d_i) = P \vee \bigcirc(d_i) = P \vee \bigcirc(P \vee \ldots \vee \bigcirc^i P) = P \vee \bigcirc P \vee \ldots \vee \bigcirc^{i+1} P$, $f(d_i)$ is also a prefix of $\Diamond P$ and $f(d_i) \in B$; when $i = \omega$, $f(d_\omega) = d_\omega \in B$. According to Scott's fixed-point induction [15], $fix_\mu(f) = \bigvee_{i \in N_0} \bigcirc^i P$ belongs to B and is a prefix of $\Diamond P$. Besides, as $fix_\mu(f)$ is the upper bound of all the elements in D and B, $fix_\mu(f)$ is the longest prefix of $\Diamond P$. Therefore, $\bigvee_{i \in N_0} \bigcirc^i P \equiv \Diamond P$. □

It is clear that $\Diamond P$ and P^+ are well-formed formulas in accordance to the syntax of PPTL. Further, by Theorem 1, index set expressions $\bigvee_{i \in N_0} \bigcirc^i P$ and $\bigvee_{i \in N_0} P^i$ are equivalent to $\Diamond P$ and P^+ respectively. Hence, we can assert that $\bigvee_{i \in N_0} \bigcirc^i P$ and $\bigvee_{i \in N_0} P^i$ are well-formed formulas.

Corollary 1. *$\bigvee_{i \in N_0} \bigcirc^i P$ and $\bigvee_{i \in N_0} P^i$ are well-formed formulas.*

Proof. This is the direct consequence of Theorem 1. □

According to Theorem 1, we can also infer that P^+ can be represented by the projection construct prj since P^+ is equivalent to $\bigvee_{i \in N_0} P^i$ and P^i is an abbreviation of

$$\underbrace{P \,;\, \ldots \,;\, P}_{i \ \ times} \equiv \underbrace{(P, \ldots, P)}_{i \ \ times} \ \mathsf{prj} \ \varepsilon$$

Thus, with techniques in this paper, $+$ can be regarded as a derived operator within PPTL.

In order to show the practical use of such index set expressions, we give an example.

Example 1. Let $P \stackrel{\text{def}}{=} \varepsilon$ in $\bigvee_{i \in N_0} \bigcirc^i P$. Then

$$\bigvee_{i \in N_0} \bigcirc^i \varepsilon \equiv \Diamond \varepsilon \equiv \mathsf{fin}$$

which claims the interval is finite and will terminate at some point. Further, we can obtain $\bigvee_{i \in N_0} \bigcirc^i \varepsilon \vee \mathsf{inf} \equiv \Diamond \varepsilon \vee \Box \mathsf{more} \equiv \mathsf{true}$.

It is interesting to consider Theorem 1 from another viewpoint. Since $\Diamond P$ can be rewritten as $P \vee \bigcirc \Diamond P$, namely $\Diamond P \equiv P \vee \bigcirc \Diamond P$, we can abstract it as a recursive equation $X \equiv P \vee \bigcirc X$ with the equality '$\equiv$' and one solution $\Diamond P$. Then $\bigvee_{i \in N_0} \bigcirc^i P$ can also be treated as a solution of the recursive equation due to its equivalence to $\Diamond P$. It is similar for the recursive equation $X \equiv P \vee P \,;\, X$, whose solution is P^+, i.e. $\bigvee_{i \in N_0} P^i$.

3.2 Generalization of $\bigvee_{i\in N_0} \bigcirc^i Q$

In this subsection, we generalize $\bigvee_{i\in N_0} \bigcirc^i Q$ to $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q$, where $P^{(0)} = \mathsf{true}$, $P^{(1)} = P, P^{(2)} = P \wedge \bigcirc P, \ldots$ and $P^{(n)} = P \wedge \bigcirc P \wedge \ldots \wedge \bigcirc^{n-1} P$ $(n \in N_0)$. In particular, when $P \equiv \mathsf{true}$, $\bigvee_{i\in N_0} \mathsf{true}^{(i)} \wedge \bigcirc^i Q$ can be exactly reduced to $\bigvee_{i\in N_0} \bigcirc^i Q$.

Further, for the recursive equation $X \equiv Q \vee P \wedge \bigcirc X$, we have:

$$\begin{aligned} X &\equiv Q \vee P \wedge \bigcirc X \\ &\equiv Q \vee P \wedge \bigcirc(Q \vee P \wedge \bigcirc X) \\ &\equiv Q \vee P \wedge \bigcirc Q \vee P \wedge \bigcirc P \wedge \bigcirc^2 X \\ &\quad \ldots \\ &\equiv Q \vee P \wedge \bigcirc Q \vee P \wedge \bigcirc P \wedge \bigcirc^2 Q \vee P \wedge \bigcirc P \wedge \bigcirc^2 P \wedge \bigcirc^3 Q \vee \ldots \end{aligned}$$

We can see that $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q$ might have something to do with the above equation, which is declared in Theorem 2.

Theorem 2. *For a recursive equation $X \equiv Q \vee P \wedge \bigcirc X$, where X, P and Q are PPTL formulas, its least fixed-point is $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q$ and its greatest fixed-point is $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \vee \Box(P \wedge \mathsf{more})$.*

Proof. At first, we prove that $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q$ and $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \vee \Box(P \wedge \mathsf{more})$ are two fixed-points of the equation $X \equiv Q \vee P \wedge \bigcirc X$ by means of simple replacement:

$$\begin{aligned} (a)\quad & Q \vee P \wedge \bigcirc(\textstyle\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q) \\ &= Q \vee \textstyle\bigvee_{i\in N_0} P^{(i+1)} \wedge \bigcirc^{i+1} Q \\ &= \textstyle\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \\ (b)\quad & Q \vee P \wedge \bigcirc(\textstyle\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \vee \Box(P \wedge \mathsf{more})) \\ &= Q \vee P \wedge \bigcirc(\textstyle\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q) \vee P \wedge \bigcirc\Box(P \wedge \mathsf{more}) \\ &= \textstyle\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \vee \Box(P \wedge \mathsf{more}) \ (P \wedge \bigcirc\Box(P \wedge \mathsf{more}) \equiv \Box(P \wedge \mathsf{more})) \end{aligned}$$

Then we respectively employ Kleene Fixed-point Theorem [15,10] and Knaster-Tarski Fixed-point Theorem [15,14] to prove $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q$ and $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \vee \Box(P \wedge \mathsf{more})$ are the least and greatest fixed-points.

Let $d_{-1} = \mathsf{false}, d_0 = Q, d_n = Q \vee P \wedge \bigcirc Q \vee P \wedge \bigcirc P \wedge \bigcirc^2 Q \vee \ldots \vee P^{(n)} \wedge \bigcirc^n Q$ $(n \in N_0), d_{\omega_1} = \bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q$ and $d_{\omega_2} = \bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \vee \Box(P \wedge \mathsf{more})$. Then we define a set $D = \{d_{-1}, d_0, \ldots, d_n, \ldots\} \cup \{d_{\omega_1}, d_{\omega_2}\}$. Further, a binary relation $\preccurlyeq$ over D is formalized as

$$d_i \preccurlyeq d_j \ \text{ if } \begin{cases} (1)\ i \le j \text{ and } i, j \in N_0 \cup \{-1\} \\ (2)\ i \in N_0 \cup \{-1\} \text{ and } j = \omega_1 \text{ or } \omega_2 \\ (3)\ i = \omega_1 \text{ and } j = \omega_1 \text{ or } \omega_2 \\ (4)\ i = \omega_2 \text{ and } j = \omega_2 \end{cases}$$

Moreover, let $g : D \to D$ be a function given below

$$g(d_i) = Q \vee P \wedge \bigcirc d_i$$

Then, for $i \in N_0 \cup \{-1\}$, $g(d_i) = Q \vee P \wedge \bigcirc(Q \vee P \wedge \bigcirc Q \vee \ldots \vee P^{(i)} \wedge \bigcirc^i Q) = Q \vee P \wedge \bigcirc Q \vee P \wedge \bigcirc P \wedge \bigcirc^2 Q \vee \ldots \vee P^{(i+1)} \wedge \bigcirc^{i+1} Q = d_{i+1}$; further, for

$i = \omega_1$ or ω_2, by (a)(b), we know that d_{ω_1} and d_{ω_2} are fixed-points of g, that is, $g(d_{\omega_1}) = d_{\omega_1}, g(d_{\omega_2}) = d_{\omega_2}$. Obviously, we have $d_i \sqcup d_j = d_i \vee d_j = d_j$ if $d_i \preccurlyeq d_j$. As a result, we can obtain the following two facts:

(1) $(D, \preccurlyeq)$ is a complete partial order and a complete lattice

$(D, \preccurlyeq)$ is a partial order, since it satisfies the properties given below:

- reflexivity: for $i \in N_0 \cup \{-1\}$, we have $d_i \preccurlyeq d_i$ due to $i \leq i$. Further, by the definition of $\preccurlyeq$, we obtain $d_{\omega_1} \preccurlyeq d_{\omega_1}, d_{\omega_2} \preccurlyeq d_{\omega_2}$.
- anti-symmetry: for $i, j \in N_0 \cup \{-1\}$, if $d_i \preccurlyeq d_j$ and $d_j \preccurlyeq d_i$, then we obtain $i \leq j$ and $j \leq i$, leading to $i = j$. Hence $d_i = d_j$. For other cases, according to the definition of $\preccurlyeq$, $d_i \preccurlyeq d_j \Longrightarrow d_j \preccurlyeq d_i$.
- transitivity: if $d_i \preccurlyeq d_j, d_j \preccurlyeq d_k$, (a) $i, j, k \in N_0 \cup \{-1\}$: by assumption, $i \leq j \leq k$, so $d_i \preccurlyeq d_k$; (b) $i \in N_0 \cup \{-1\}, j \in N_0 \cup \{-1, \omega_1, \omega_2\}, k \in \{\omega_1, \omega_2\}$: according to case (2) in the definition of $\preccurlyeq$, we can acquire $d_i \preccurlyeq d_k$; (c) $i, j, k \in \{\omega_1, \omega_2\}$, it is clear that $d_i \preccurlyeq d_k$ by the cases (3) and (4) in the definition of $\preccurlyeq$.

Furthermore, for any non-empty subset $S = \{d_{i_1}, \ldots, d_{i_n}\}$ of D, we consider the following cases:

(a) S includes neither d_{ω_1} nor d_{ω_2}:

In this case, if S is finite, as $d_i \sqcup d_j = d_j$ $(i \leq j)$, we obtain the least upper bound $d_{i_n} \in D$ with i_n the biggest index in S; otherwise, S is infinite, there exists a least upper bound $\bigsqcup_{i_n \in N_0} d_{i_n} = \bigvee_{i_n \in N_0} d_{i_n} = (Q \vee P \wedge \bigcirc Q \vee \ldots \vee P^{(i_1)} \wedge \bigcirc^{i_1} Q) \vee (Q \vee P \wedge \bigcirc Q \vee \ldots \vee P^{(i_2)} \wedge \bigcirc^{i_2} Q) \vee \ldots = Q \vee P \wedge \bigcirc Q \vee P \wedge \bigcirc P \wedge \bigcirc^2 Q \vee \ldots = \bigvee_{i \in N_0} P^{(i)} \wedge \bigcirc^i Q = d_{\omega_1}$, which evidently belongs to D.

(b) S involves d_{ω_1} or d_{ω_2}:

If d_{ω_2} is contained in S, it is the least upper bound in S; otherwise, d_{ω_1} is the least upper bound in S.

Thus, $(D, \preccurlyeq)$ is a complete lattice, which is also a complete partial order.

(2) g is a continuous function

Suppose that $d_i \preccurlyeq d_j$, then (a) when $i, j \in N_0 \cup \{-1\}$: $i \leq j$, so $i + 1 \leq j + 1$. As a result, $g(d_i) = d_{i+1} \preccurlyeq d_{j+1} = g(d_j)$; (b) when $i \in N_0 \cup \{-1\}$ and $j = \omega_1$ or ω_2, $g(d_i) = d_{i+1} \preccurlyeq d_{\omega_t} = g(d_{\omega_t})$ $(t = 1, 2)$; (c) when $i, j \in \{\omega_1, \omega_2\}$, $g(d_i) = d_i \preccurlyeq d_j = g(d_j)$. Hence, g is monotonic.

Further, for an arbitrary ω-chain in D, $d_{i_1} \preccurlyeq d_{i_2} \preccurlyeq \ldots \preccurlyeq d_{i_n} \preccurlyeq \ldots$, if there exists an element d_{i_n} such that $d_{i_n} \preccurlyeq d_{i_n} \preccurlyeq \ldots$, it is obvious that d_{i_n} is the least upper bound of this ω-chain. Thus, we acquire

$$g(\bigsqcup_{i_n \in N_0} d_{i_n}) = g(d_{i_n}) = d_{i_n+1} = d_{i_n} \sqcup d_{i_n+1} = (\bigsqcup_{i_n \in N_0} d_{i_n}) \sqcup d_{i_n+1} = \bigsqcup_{i_n \in N_0} d_{i_n+1} = \bigsqcup_{i_n \in N_0} g(d_{i_n})$$

Otherwise,

$$\bigsqcup_{i_n \in N_0} g(d_{i_n}) = \bigsqcup_{i_n \in N_0} d_{i_n+1} = d_0 \sqcup \bigsqcup_{i_n \in N_0} d_{i_n+1} = \bigsqcup_{i_n \in N_0} d_{i_n} = d_{\omega_1} = g(d_{\omega_1}) = g(\bigsqcup_{i_n \in N_0} d_{i_n})$$

Therefore, g is a continuous function. Based on these, we can prove:

1. $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q$ is the least fixed-point

Since $(D, \preccurlyeq)$ is a complete partial order, whose bottom element is false, and g is a continuous function, by Kleene fixed-point theorem, there exists a least fixed-point:

$$\begin{aligned} fix_\mu(g) = \bigsqcup_{n\in N_0} g^n(d_{-1}) = \bigsqcup_{n\in N_0} d_{n-1} &= d_{-1} \sqcup \bigsqcup_{n\in N_0} d_n \\ &= \bigsqcup_{n\in N_0} d_n \\ &= d_{\omega_1} = \bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \end{aligned}$$

2. $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \vee \Box(P \wedge \mathsf{more})$ is the greatest fixed-point

Let $\{x \in D | x \preccurlyeq g(x)\}$ be the set of post fixed-points of g. As $d_i \preccurlyeq d_{i+1} = g(d_i)$ $(i \in N_0 \cup \{-1\})$ and $d_{\omega_j} \preccurlyeq d_{\omega_j} = g(d_{\omega_j})$ $(j = 1, 2)$, we obtain $\{x \in D | x \preccurlyeq g(x)\} = D$. In other words, all the elements in D are post fixed-points of g. Further, $(D, \preccurlyeq)$ is a complete lattice, whose bottom element is false, and g is monotonic. According to Knaster-Tarski fixed-point theorem, the greatest fixed-point is:

$$\begin{aligned} fix_\nu(g) = \bigsqcup\{x \in D | x \preccurlyeq g(x)\} &= \bigsqcup D \\ &= d_{\omega_2} \\ &= \bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \vee \Box(P \wedge \mathsf{more}) \end{aligned}$$

□

Corollary 2. *For a recursive equation $X \equiv Q \vee \bigcirc X$, where X and Q are PPTL formulas, its least fixed-point is $\Diamond Q$ and its greatest fixed-point is $\Diamond Q \vee \Box\mathsf{more}$.*

Proof. In Theorem 2, let $P \equiv \mathsf{true}$. Then for the equation $X \equiv Q \vee \bigcirc X$, its least fixed-point is $\bigvee_{i\in N_0} (\mathsf{true})^{(i)} \wedge \bigcirc^i Q \equiv \bigvee_{i\in N_0} \bigcirc^i Q$. Further, by Theorem 1, $\bigvee_{i\in N_0} \bigcirc^i Q \equiv \Diamond Q$, so $\Diamond Q$ is the least fixed-point of the equation $X \equiv Q \vee \bigcirc X$. Moreover, its greatest fixed-point is $\bigvee_{i\in N_0} (\mathsf{true})^{(i)} \wedge \bigcirc^i Q \vee \Box(\mathsf{true} \wedge \mathsf{more}) \equiv \Diamond Q \vee \Box\mathsf{more}$. □

In fact, Corollary 2 is a special case of Theorem 2 as well as the equation $X \equiv Q \vee \bigcirc X$ is an instance of $X \equiv Q \vee P \wedge \bigcirc X$ with $P \equiv \mathsf{true}$ but has well-formed formulas as its least and greatest fixed-points. In addition, Theorem 2 also tells us that there are at least two fixed-points for the equation $X \equiv Q \vee P \wedge \bigcirc X$. Actually, $X \equiv Q \vee P \wedge \bigcirc X$ has and only has two fixed-points, namely the least and greatest fixed-points, which is confirmed by Theorem 3.

Theorem 3. *The recursive equation $X \equiv Q \vee P \wedge \bigcirc X$ has and only has two solutions, i.e. $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q$ and $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \vee \Box(P \wedge \mathsf{more})$.*

Proof. It is clear that $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q$ and $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \vee \Box(P \wedge \mathsf{more})$ are two fixed-points of $X \equiv Q \vee P \wedge \bigcirc X$ by Theorem 2. Further, we prove the equation only has two fixed-points. We assume that there exists a third solution R such that $R \equiv Q \vee P \wedge \bigcirc R$. Since $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q$ is the least fixed-point, $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \preccurlyeq R$. Further, according to the proof of Theorem 2, we can acquire $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \sqcup R =$

$\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \vee R = R$. Thus, R must be in the form of $\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \vee R'$. Therefore, we have,

$$\begin{aligned} & \bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \vee R' \\ \equiv\ & Q \vee P \wedge \bigcirc(\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \vee R') \\ \equiv\ & Q \vee P \wedge \bigcirc(\bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q) \vee P \wedge \bigcirc R' \\ \equiv\ & \bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \vee P \wedge \bigcirc R' \end{aligned}$$

Accordingly, we can infer $R' \equiv P \wedge \bigcirc R'$, which can only be satisfied when $R' \equiv \mathsf{false}$ or $R' \equiv \Box(P \wedge \mathsf{more})$ within PPTL. As a result, if $R' \equiv \mathsf{false}$, $R \equiv \bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q$ while if $R' \equiv \Box(P \wedge \mathsf{more})$, $R \equiv \bigvee_{i\in N_0} P^{(i)} \wedge \bigcirc^i Q \vee \Box(P \wedge \mathsf{more})$. Hence, the equation only has two fixed-points. □

3.3 Examples

To intuitively understand the above theorems, we present some examples below.

Example 2. Let $P \stackrel{\text{def}}{=} R$ and $Q \stackrel{\text{def}}{=} R \wedge \varepsilon$. Then $X \equiv Q \vee P \wedge \bigcirc X$ can be instantiated as

$$X \equiv R \wedge \varepsilon \vee R \wedge \bigcirc X \tag{3.3.1}$$

According to Theorem 2, we can respectively obtain the least and greatest fixed-points of the equation (3.3.1) as

$$\bigvee_{i\in N_0} R^{(i)} \wedge \bigcirc^i (R \wedge \varepsilon) \quad \text{and} \quad \bigvee_{i\in N_0} R^{(i)} \wedge \bigcirc^i (R \wedge \varepsilon) \vee \Box(R \wedge \mathsf{more})$$

where the greatest fixed-point claims that R always holds either over an interval with the length i or over an infinite interval. On the other hand, since the logical law

$$\Box R \equiv R \wedge \varepsilon \vee R \wedge \bigcirc \Box R$$

can be satisfied, we can infer that $\Box R$ is one solution of the equation (3.3.1), and must be equivalent to either the least or the greatest fixed-points by Theorem 3. In accordance with the meaning of $\Box R$, we can see that the greatest fixed-point exactly characterizes $\Box R$ and acquire the following:

$$\bigvee_{i\in N_0} R^{(i)} \wedge \bigcirc^i (R \wedge \varepsilon) \vee \Box(R \wedge \mathsf{more}) \equiv \Box R$$

which convinces us $\bigvee_{i\in N_0} R^{(i)} \wedge \bigcirc^i (R \wedge \varepsilon)$ is well-formed.

Example 3. Let $P \stackrel{\text{def}}{=} \mathsf{true}$ and $Q \stackrel{\text{def}}{=} R \wedge \varepsilon$. Then $X \equiv Q \vee P \wedge \bigcirc X$ can be instantiated as

$$X \equiv R \wedge \varepsilon \vee \bigcirc X \tag{3.3.2}$$

whose least and greatest fixed-points respectively are:

$$\bigvee_{i\in N_0} \bigcirc^i (R \wedge \varepsilon) \equiv \Diamond(R \wedge \varepsilon) \quad \text{and} \quad \bigvee_{i\in N_0} \bigcirc^i (R \wedge \varepsilon) \vee \Box\mathsf{more} \equiv \Diamond(R \wedge \varepsilon) \vee \Box\mathsf{more}$$

Particularly, the greatest fixed-point states that R will hold and terminate at some point over a finite interval or the interval is infinite. Further, we have the logical law

$$\mathsf{fin}(R) \equiv R \wedge \varepsilon \vee \bigcirc \mathsf{fin}(R)$$

so $\mathsf{fin}(R)$ is one solution of the equation (3.3.2) and equivalent to the greatest fixed-point by its meaning. Thus, we can obtain:

$$\mathsf{fin}(R) \equiv \bigvee_{i \in N_0} \bigcirc^i (R \wedge \varepsilon) \vee \Box \mathsf{more} \equiv \Diamond (R \wedge \varepsilon) \vee \Box \mathsf{more}$$

Example 4. Let $P \stackrel{\text{def}}{=} R$ and $Q \stackrel{\text{def}}{=} \varepsilon$. Then $X \equiv Q \vee P \wedge \bigcirc X$ can be instantiated as

$$X \equiv \varepsilon \vee R \wedge \bigcirc X \tag{3.3.3}$$

whose least and greatest fixed-points can be attained as:

$$\bigvee_{i \in N_0} R^{(i)} \wedge \bigcirc^i \varepsilon \quad \text{and} \quad \bigvee_{i \in N_0} R^{(i)} \wedge \bigcirc^i \varepsilon \vee \Box (R \wedge \mathsf{more})$$

In particular, the greatest fixed-point tells us that R is true at every state over an infinite interval or over a finite interval with ignoring the final state. Further, the logical law

$$\mathsf{keep}(R) \equiv \varepsilon \vee R \wedge \bigcirc \mathsf{keep}(R)$$

holds and implies $\mathsf{keep}(R)$ is one solution of the equation (3.3.3). Since $\mathsf{keep}(R)$ precisely specifies the meaning of the greatest fixed-point, we have:

$$\mathsf{keep}(R) \equiv \bigvee_{i \in N_0} R^{(i)} \wedge \bigcirc^i \varepsilon \vee \Box (R \wedge \mathsf{more})$$

which makes $\bigvee_{i \in N_0} R^{(i)} \wedge \bigcirc^i \varepsilon$ well-formed.

Example 5. Let $P \stackrel{\text{def}}{=} \neg R$ and $Q \stackrel{\text{def}}{=} R \wedge \varepsilon$. Then $X \equiv Q \vee P \wedge \bigcirc X$ can be instantiated as

$$X \equiv R \wedge \varepsilon \vee \neg R \wedge \bigcirc X \tag{3.3.4}$$

By Theorem 2, its least and greatest fixed-points respectively are:

$$\bigvee_{i \in N_0} (\neg R)^{(i)} \wedge \bigcirc^i (R \wedge \varepsilon) \quad \text{and} \quad \bigvee_{i \in N_0} (\neg R)^{(i)} \wedge \bigcirc^i (R \wedge \varepsilon) \vee \Box (\neg R \wedge \mathsf{more})$$

where the greatest fixed-point asserts that R is only true at the final state over a finite interval or $\neg R$ always holds over an infinite interval. Moreover, we have known that

$$\mathsf{halt}(R) \equiv R \wedge \varepsilon \vee \neg R \wedge \bigcirc \mathsf{halt}(R)$$

which indicates $\mathsf{halt}(R)$ is one solution of the equation (3.3.4). As $\mathsf{halt}(R)$ exactly expresses the greatest fixed-point, we can get:

$$\mathsf{halt}(R) \equiv \bigvee_{i \in N_0} (\neg R)^{(i)} \wedge \bigcirc^i (R \wedge \varepsilon) \vee \Box (\neg R \wedge \mathsf{more})$$

which further convinces us $\bigvee_{i \in N_0} (\neg R)^{(i)} \wedge \bigcirc^i (R \wedge \varepsilon)$ is well-formed.

Example 6. Let $P \stackrel{\text{def}}{=} \mathsf{true}$ and $Q \stackrel{\text{def}}{=} \varepsilon$. Then $X \equiv Q \vee P \wedge \bigcirc X$ can be instantiated as

$$X \equiv \varepsilon \vee \bigcirc X \tag{3.3.5}$$

Further, the least and greatest fixed-points can be acquired as

$$\bigvee_{i \in N_0} \mathsf{true}^{(i)} \wedge \bigcirc^i \varepsilon \equiv \bigvee_{i \in N_0} \bigcirc^i \varepsilon \text{ and } \bigvee_{i \in N_0} \mathsf{true}^{(i)} \wedge \bigcirc^i \varepsilon \vee \Box \mathsf{more} \equiv \bigvee_{i \in N_0} \bigcirc^i \varepsilon \vee \Box \mathsf{more}$$

which respectively says that the interval is finite and the interval is finite or infinite. On the other hand, we have:

$$\mathsf{fin} \equiv \Diamond \varepsilon \equiv \varepsilon \vee \bigcirc \Diamond \varepsilon \text{ and } \mathsf{true} \equiv \varepsilon \vee \bigcirc \mathsf{true}$$

which suggests that $\Diamond \varepsilon$ and true are the solutions of the equation (3.3.5). Hence, according to their meanings, we can attain:

$$\bigvee_{i \in N_0} \bigcirc^i \varepsilon \equiv \Diamond \varepsilon \equiv \mathsf{fin} \text{ and } \bigvee_{i \in N_0} \bigcirc^i \varepsilon \vee \Box \mathsf{more} \equiv \mathsf{true}$$

which is consistent with Example 1.

4 Representation of 'U' and 'W' of PLTL within PPTL

Linear Temporal Logic (LTL) [13] is a well-known temporal logic, which is based on a linear-time perspective and often defined over an infinite path (i.e. an infinite interval). Propositional LTL (PLTL) is a propositional subset of LTL and has been widely used in practice. In PLTL, the most prominent operators are 'U' (strong until) and 'W' (weak until), where 'W' is a weak version of 'U'. Their intuitive semantics are shown in Figure 1(a) and (b) respectively and more details can be found in [2]. Except U and W operators, other operators of PLTL can be directly formalized over an infinite interval in PPTL. In this section, we employ techniques proposed in Section 3 to equivalently express 'U' and 'W' constructs within PPTL.

In PLTL, the following laws have been proved:

$$\begin{array}{ll} P \,\mathsf{U}\, Q \equiv (P \wedge \neg Q) \,\mathsf{U}\, Q & P \,\mathsf{U}\, Q \equiv Q \vee P \wedge \bigcirc (P \,\mathsf{U}\, Q) \\ P \,\mathsf{W}\, Q \equiv (P \wedge \neg Q) \,\mathsf{W}\, Q & P \,\mathsf{W}\, Q \equiv Q \vee P \wedge \bigcirc (P \,\mathsf{W}\, Q) \\ P \,\mathsf{W}\, Q \equiv (P \,\mathsf{U}\, Q) \vee \Box P & \neg \bigcirc P \equiv \bigcirc \neg P \end{array}$$

Hence, $P \,\mathsf{U}\, Q$ can be reduced as follows:

$$\begin{array}{l} P \,\mathsf{U}\, Q \equiv (P \wedge \neg Q) \,\mathsf{U}\, Q \\ \quad \equiv Q \vee (P \wedge \neg Q) \wedge \bigcirc ((P \wedge \neg Q) \,\mathsf{U}\, Q) \\ \quad \equiv Q \vee (P \wedge \neg Q) \wedge \bigcirc (Q \vee (P \wedge \neg Q) \wedge \bigcirc ((P \wedge \neg Q) \,\mathsf{U}\, Q)) \\ \quad \equiv Q \vee (P \wedge \neg Q) \wedge \bigcirc Q \vee (P \wedge \neg Q) \wedge \bigcirc (P \wedge \neg Q) \wedge \bigcirc^2 ((P \wedge \neg Q) \,\mathsf{U}\, Q) \\ \quad \equiv Q \vee (P \wedge \neg Q) \wedge \bigcirc Q \vee (P \wedge \neg Q) \wedge \bigcirc (P \wedge \neg Q) \wedge \bigcirc^2 Q \vee \ldots \end{array}$$

From the above, we find that the recursive equation of $P \,\mathsf{U}\, Q$ can be treated as the form of $X \equiv Q \vee P \wedge \neg Q \wedge \bigcirc X$ $(\star\star)$ with one solution $P \,\mathsf{U}\, Q$. Further, by Theorem 2 and

the semantics of $P \mathsf{U} Q$, we can obtain the least fixed-point $\bigvee_{i\in N_0}(P \wedge \neg Q)^{(i)} \wedge \bigcirc^i Q$ of the equation $(\star\star)$, which corresponds to $P \mathsf{U} Q$. In other words, $P \mathsf{U} Q$ is equivalent to $\bigvee_{i\in N_0}(P \wedge \neg Q)^{(i)} \wedge \bigcirc^i Q$. However, formulas in PPTL can be interpreted over both infinite and finite intervals whereas formulas in PLTL can only be satisfied by infinite paths. Therefore, in order to force a PPTL formula to hold just over an infinite interval, an additionally PPTL formula $\Box$more is needed. Thus, we can equivalently represent $P \mathsf{U} Q$ within PPTL as follows:

$$P \mathsf{U} Q \stackrel{\text{def}}{=} (\bigvee_{i\in N_0}(P \wedge \neg Q)^{(i)} \wedge \bigcirc^i Q) \wedge \Box\mathsf{more} \quad (\star\star\star)$$

Similar to $P \mathsf{U} Q$, the recursive equation of $P \mathsf{W} Q$ is $P \mathsf{W} Q \equiv (P \wedge \neg Q) \mathsf{W} Q \equiv Q \vee (P \wedge \neg Q) \wedge \bigcirc((P \wedge \neg Q) \mathsf{W} Q)$ and also in the form of the equation $(\star\star)$. Further, according to the semantics of $P \mathsf{W} Q$ and by Theorem 2, $P \mathsf{W} Q$ is equivalent to the greatest fixed-point $\bigvee_{i\in N_0}(P \wedge \neg Q)^{(i)} \wedge \bigcirc^i Q \vee \Box(P \wedge \neg Q \wedge \mathsf{more})$ of the equation $(\star\star)$. Therefore, with the requirement of an infinite interval, we have the following:

$$\begin{aligned} P \mathsf{W} Q &\stackrel{\text{def}}{=} (\bigvee_{i\in N_0}(P \wedge \neg Q)^{(i)} \wedge \bigcirc^i Q \vee \Box(P \wedge \neg Q \wedge \mathsf{more})) \wedge \Box\mathsf{more} \\ &\equiv (\bigvee_{i\in N_0}(P \wedge \neg Q)^{(i)} \wedge \bigcirc^i Q) \wedge \Box\mathsf{more} \vee \Box(P \wedge \neg Q \wedge \mathsf{more}) \\ &\equiv P \mathsf{U} Q \vee \Box(P \wedge \neg Q \wedge \mathsf{more}) \end{aligned}$$

With techniques presented in this paper, we can see that $P \mathsf{U} Q$ is the least fixed-point while $P \mathsf{W} Q$ is the greatest fixed-point of the equation $(\star\star)$, which is coherent with that in [2].

Example 7. We consider a LTL formula $\bigcirc^3 p \mathsf{U} \bigcirc q$, where p, q are atomic propositions. Three possible paths are shown in Figure 1(c)(1-3). Further, according to the equation $(\star\star\star)$, an equivalent PPTL formula can be acquired as $(\bigvee_{i\in N_0}(\bigcirc^3 p \wedge \neg\bigcirc q)^{(i)} \wedge \bigcirc^i \bigcirc q) \wedge \Box$more. Next we reduce the PPTL formula to obtain the three paths for showing the correctness of the equation $(\star\star\star)$.

Firstly, we know that

$$\begin{aligned} &(\bigvee_{i\in N_0}(\bigcirc^3 p \wedge \neg \bigcirc q)^{(i)} \wedge \bigcirc^i \bigcirc q) \wedge \Box\mathsf{more} \\ \equiv\ &(\bigvee_{i\in N_0}(\bigcirc^3 p \wedge \bigcirc\neg q)^{(i)} \wedge \bigcirc^i \bigcirc q) \wedge \Box\mathsf{more} \\ \equiv\ &\bigcirc q \wedge \Box\mathsf{more} \vee (\bigcirc^3 p \wedge \bigcirc\neg q) \wedge \bigcirc \bigcirc q \wedge \Box\mathsf{more} \vee \\ &(\bigcirc^3 p \wedge \bigcirc\neg q) \wedge \bigcirc(\bigcirc^3 p \wedge \bigcirc\neg q) \wedge \bigcirc^2 \bigcirc q \wedge \Box\mathsf{more} \vee \ldots \end{aligned}$$

Then $\bigcirc q \wedge \Box$more characterizes path (1), $(\bigcirc^3 p \wedge \bigcirc\neg q) \wedge \bigcirc^2 q \wedge \Box$more describes path (2) and $(\bigcirc^3 p \wedge \bigcirc\neg q) \wedge \bigcirc(\bigcirc^3 p \wedge \bigcirc\neg q) \wedge \bigcirc^3 q \wedge \Box$more specifies path (3). We merely reduce $(\bigcirc^3 p \wedge \bigcirc\neg q) \wedge \bigcirc(\bigcirc^3 p \wedge \bigcirc\neg q) \wedge \bigcirc^3 q \wedge \Box$more to illustrate how to get the relevant path and others can be obtained in a similar manner.

$$\begin{aligned} &(\bigcirc^3 p \wedge \bigcirc\neg q) \wedge \bigcirc(\bigcirc^3 p \wedge \bigcirc\neg q) \wedge \bigcirc^3 q \wedge \Box\mathsf{more} \\ \equiv\ &\bigcirc(\bigcirc^2 p \wedge \neg q \wedge \bigcirc^3 p \wedge \bigcirc\neg q \wedge \bigcirc^2 q \wedge \Box\mathsf{more}) \end{aligned}$$

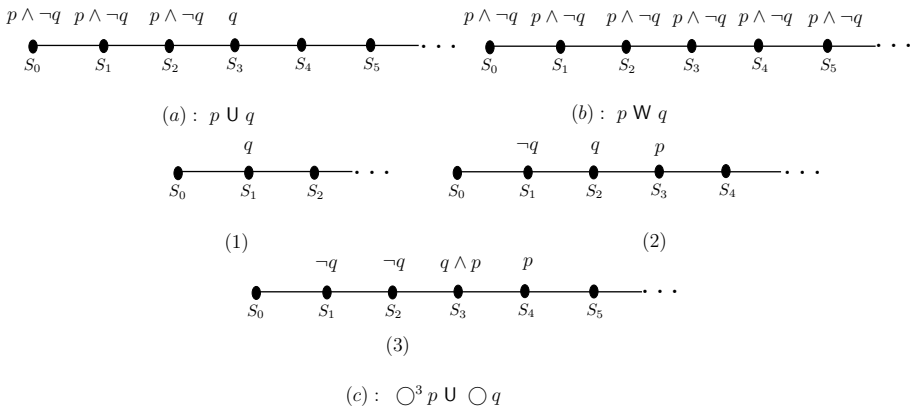


Fig. 1. Intuitive meaning of $p \mathsf{U} q$ and $p \mathsf{W} q$ and some models of $\bigcirc^3 p \mathsf{U} \bigcirc q$

Thus true holds at state s_0. Next, at state s_1, we continue to reduce

$$\bigcirc^2 p \wedge \neg q \wedge \bigcirc^3 p \wedge \bigcirc \neg q \wedge \bigcirc^2 q \wedge \Box \mathsf{more} \equiv \neg q \wedge \bigcirc(\bigcirc p \wedge \bigcirc^2 p \wedge \neg q \wedge \bigcirc q \wedge \Box \mathsf{more})$$

From this, we can see $\neg q$ holds at state s_1. Further, at state s_2,

$$\bigcirc p \wedge \bigcirc^2 p \wedge \neg q \wedge \bigcirc q \wedge \Box \mathsf{more} \equiv \neg q \wedge \bigcirc(p \wedge \bigcirc p \wedge q \wedge \Box \mathsf{more})$$

Therefore, $\neg q$ is satisfied by state s_2. At state s_3, we go on reducing and get below:

$$p \wedge \bigcirc p \wedge q \wedge \Box \mathsf{more} \equiv p \wedge q \wedge \bigcirc(p \wedge \Box \mathsf{more})$$

Then $p \wedge q$ holds at state s_3. Subsequently, at state s_4,

$$p \wedge \Box \mathsf{more} \equiv p \wedge \bigcirc(\Box \mathsf{more})$$

which makes p hold at state s_4 and all the successive states over an infinite path be satisfied by $true$ (i.e arbitrary propositions). Hence, we attain the path (3).

5 Conclusion

This paper investigated some fixed-point issues within PPTL. Particularly, we give two kinds of index set expressions $\bigvee_{i \in N_0} \bigcirc^i P$ and $\bigvee_{i \in N_0} P^i$, which are formed by applying rules of the PPTL syntax infinitely many times. Further, we proved that these formulas expressed by index set expressions are still well-formed PPTL formulas. Moreover, $\bigvee_{i \in N_0} \bigcirc^i Q$ is generalized to $\bigvee_{i \in N_0} P^{(i)} \wedge \bigcirc^i Q$ and the least and greatest fixed-points of the equation $X \equiv Q \vee P \wedge \bigcirc X$ are explored. In addition, the operators 'U' and 'W' in PLTL are equivalently represented within PPTL in terms of $\bigvee_{i \in N_0} P^{(i)} \wedge \bigcirc^i Q$.

In this paper, we only demonstrate some instances of the index set expression $\bigvee_{i \in N_0} P^{(i)} \wedge \bigcirc^i Q$ with specific formulas as P and Q are well-formed PPTL formulas but do

not give its equivalent generic well-formed formulas. As a challenge, we will attempt to find out its concrete well-formed formula in the near future. Further, we will work out the conditions for the solutions of $X \equiv Q \vee P \wedge \bigcirc X$, so that we know when X takes the least fixed-point and when X takes the greatest fixed-point. Moreover, formulas under investigation possess a common feature that during their recursive rewriting, only one or two formulas appear repeatedly. For example, in $\bigvee_{i \in N_0} \bigcirc^i P$, P occurs iteratively for infinite number of times, and so do P and Q in $\bigvee_{i \in N_0} P^{(i)} \wedge \bigcirc^i Q$. However, for these formulas such as $\bigvee_{i \in N_0} \bigcirc^i P_i$, where P_i's might be different for distinct i, whether or not they are well-formed is an open question. We will study the problem in the future.

References

1. Clarke, E.M., Allen Emerson, E.: Design and synthesis of synchronization skeletons using branching timed temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
2. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Massachusetts (2000)
3. Duan, Z.: An Extended Interval Temporal Logic and a Framing Technique for Temporal Logic Programming. PhD thesis, University of Newcastle Upon Tyne (May 1996)
4. Duan, Z.: Temporal Logic and Temporal Logic Programming Language. Science Press, Beijing (2006)
5. Duan, Z., Koutny, M., Holt, C.: Projection in temporal logic programming. In: Pfenning, F. (ed.) LPAR 1994. LNCS (LNAI), vol. 822, pp. 333–344. Springer, Heidelberg (1994)
6. Duan, Z., Tian, C., Zhang, L.: A decision procedure for propositional projection temporal logic with infinite models. Acta Informatica 45, 43–78 (2008)
7. Duan, Z., Zhang, N., Koutny, M.: A complete proof system for propositional projection temporal logic. Theoretical Computer Science (2012), doi:10.1016/j.tcs.2012.01.026
8. Gordon, M.: Twenty years of theorem proving for hols past, present and future. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 1–5. Springer, Heidelberg (2008)
9. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of ACM 12, 576–583 (1969)
10. Kleene, S.C.: Mathematical logic. John Wiley (1967)
11. Manna, Z., Pnueli, A.: Temporal Logic of Reactive and Concurrent Systems. Springer, Berlin (1992)
12. Moszkowski, B.C.: Executing Temporal Logic Programs. PhD thesis, Cambridge Uniersity (1986)
13. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annal IEEE Symp. on Foudations of Computer Science, pp. 46–57. IEEE Computer Society (1977)
14. Tarski, A.: A lattic-theoretical fixpoint theorem and its application. Pacific Journal of Mathematics 5, 285–309 (1955)
15. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. The MIT Press, Cambridge (1993)

The Value-Passing Calculus

Yuxi Fu

BASICS, Department of Computer Science, Shanghai Jiaotong University
MOE-MS Key Laboratory for Intelligent Computing and Intelligent Systems

Abstract. A value-passing calculus is a process calculus in which the contents of communications are values chosen from some data domain, and the propositions appearing in the conditionals are formulas constructed from a logic. Previous studies treat the domain models, as well as the logic theories, as unspecified oracles. The open-ended approach leaves open some fundamental issues unanswered. The paper provides a more formal account of the value-passing calculi. The new treatment is self-contained in that the logic theory a value-passing calculus refers to is formally defined. A value-passing calculus consists of a complete first order theory with an operational model that makes use of the terms and the boolean expressions of the theory. A systematic investigation into the theory of the value-passing calculi is carried out. A particular value-passing calculus, $\mathbb{VPC}$, is shown to be the least expressive among all Turing complete value-passing calculi.

1 Introduction

Process calculus offers one approach to study interactions between computing objects. The process models can be classified by the type of the entities exchanged over interactions. The pioneering process calculus, the CCS of Milner [Mil89a], abstracts away the contents of communications. For this reason, it serves as a benchmark model for process. Although the pure CCS falls short of being a very interesting model from the point of view of expressiveness [Fu12b], the basic theory of CCS does generalize to many process calculi. The value-passing CCS [Mil89a] adds to CCS the capacity to pass data values between processes. In addition to the simple mechanism of synchronization, communications of values render it possible to control the interaction flow by testing the received data values. This additional control power significantly enhances the expressive power of the value-passing calculi. The name-passing calculus of Milner, Parrow and Walker [MPW92], the π-calculus, adopts the policy that the messages sent and received in communications can only be channel names. The exclusive focus on the names has achieved both simplicity and expressiveness. It is difficult to extend the name-passing mechanism to get a strictly stronger model. The process-passing calculi, or the higher order calculi [San93, Tho89, Tho93, Tho95], have typically processes as the contents of communications. This seemingly powerful communication mechanism turns out to be much less expressive than the

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 166–195, 2013.

π-calculus [Fu12b]. To make use of what have been received through communications, variables (value variables, name variables, process variables) have to be introduced to act as placeholders.

From a logic point of view, the name-passing calculi and the process-passing calculi are preferable since they are closed models in the sense that the syntax and the semantics of these calculi are independent of any models or logics. In contrary the value-passing calculi, with strong motivation from practice, are distinguished by the fact that they must refer to an 'oracle', be it a domain model or a logic theory. The traditional treatment to the value-passing calculi are not self-contained. The attentions have largely been on the process aspect of the story. The oracles have never been formally defined. The under-specification of the value domain is not welcome from a foundational viewpoint, nor is it really useful in practice.

The lightweight treatment of the oracle models/logics is an obstacle to both theoretical study and application. We mention three immediate consequences.

1. Deep theoretical investigations are inevitably hindered by the open-ended approach. For example it is not always possible to compare the expressiveness of a value-passing calculus to another concurrent model. An encoding of the former into the latter would require that the value terms and the logic expressions be fully specified.
2. For the same reason there is no way to implement a value-passing calculus, no matter what is meant by an implementation.
3. An equivalence checking algorithm is out of the question since the existence of such an algorithm depends on the algorithmic aspect of the oracle model and/or the oracle logic, which is not available in the open-ended framework.

Apart from these problems, there are also a number of related subtleties that have to be taken into account when designing a value-passing calculus. Let's illustrate these points by scrutinizing the process $M(x)$ defined by the following recursive equality.

$$M(x) = \mathit{if}\, \varphi(x)\ \mathit{then}\, \overline{a}(f(t))\ \mathit{else}\, M(x+1). \tag{1}$$

There are at least five questions one may ask about the process defined in (1). The first is concerned with the nature of the oracle. Where are $\varphi(x)$ and t coming from? There are basically two answers.

1. The first answer is model theoretical. The value term t and the logical expression $\varphi(x)$ are constructed from the elements of the universe of a model, the functions and the relations on the universe, and the variables that range over the universe. The logical expression φ must be evaluated in the model before process (1) fires an action. If φ contains free variables, the evaluation is done with respect to an assignment. Under the model theoretical interpretation, one expects that the process expression $\mathit{if}\, y = y\ \mathit{then}\, P\ \mathit{else}\, Q$ can be immediately put into action. On the other hand the behavior of the process expression $\mathit{if}\, y = z\ \mathit{then}\, P\ \mathit{else}\, Q$ depends on particular assignments.

2. The second answer is proof theoretical. The value term t is defined inductively from a vocabulary and the logical expression φ is legitimate in a first order theory on top of the vocabulary. The process expression in (1) can fire if φ is a theorem of the theory.

In principle, a proof theoretical approach to oracle design should definitely be preferred. The point is that all implementations of an oracle are proof theoretical in nature. As the Incompleteness Theorem of Gödel [G$\ddot{3}$1] tells us, the set of the statements true in a model is far from being recursively enumerable. But an implemented system can only generate a recursively enumerable set of theorems. The proof theoretical approach fits very well with the operational nature of process calculi.

The second question is about the expressiveness of the logical expressions appearing in the value-passing processes. Since the set of the theorems of a first order theory is typically creative [Cut80], there is in general no effective procedure to decide the theoremhood of a formula. If a theory is reasonably expressive, there exists some first order logical expression ψ such that neither ψ nor $\neg\psi$ is provable. This is definitely unacceptable from a programming point of view. To avoid the embarrassment caused by the Incompleteness Theorem, one looks for decidable theories in which a sentence is either provably true or provably false. In implementation one actually asks for more than decidability. It has been shown that validity checking for a decidable theory could be super exponential in complexity [FR74, Opp78]. So normally the logical expressions admitted in a value-passing calculus are confined to the quantifier free formulas. But this restriction does not entirely eliminate the problem. If φ contains the free variables $x_1, \ldots, x_n$, then proving φ is equivalent to proving $\forall x_1 \ldots \forall x_n . \varphi$. There are situations, for example in equivalence checking algorithm, where formulas with free variables must be dealt with. This suggests to look for first order theories that are considerably weaker than say Peano Arithmetic.

The third question is about the expressive power of the value terms admitted in a value-passing calculus. Our value-passing calculus would be too strong if the f appearing in (1) could be a non-computable function. The Church-Turing Thesis asserts that all the functions definable in a value-passing calculus are computable functions if the functions produced by the oracles are computable. It would be reasonable to disown those oracles that are capable of delivering non-computable functions. Now here is the twist, if all the recursive functions can be defined within a value-passing calculus, is it necessary to have an oracle that produces functions short-cutting the role of the definable functions? A negative answer would imply that the oracle should only supply constructors for the value terms; it should not introduce any functions that compute on the value terms.

The fourth question is about the functional separation between the calculi and the oracles. The standard semantics of the value-passing calculi demands that the value term $f(t)$ must be calculated to a canonical value before it is exported at the name a. This additional calculating machinery is not very appealing from the point of view of an interaction model. In process calculi all calculations should be achieved by interactions. In other words, the procedure of the calculation

should be explicitly specified in a process, not implicitly done by an oracle. It is interesting to notice that the negative answer to the previous question is also an answer to the present question since it trivializes the issue.

The fifth question is about the level of abstraction of the value-passing calculi. If n is the least natural number such that $\varphi(n)$ holds according to an oracle, then $M(0)$ emits $f(t)$ at the channel named a; otherwise $M(0)$ is inactive. If $M(0)$ ever interacts, it need to consult the oracle for a finite number of times. If $M(0)$ never interacts, it must consult the oracle to evaluate $\varphi(0), \varphi(1), \varphi(2), \ldots$ consecutively in a non-stop fashion. This phenomenon is familiar to higher order programming languages, it is however alien to process calculi. The process $M(0)$ has abstracted away too many computational and interactional activities, the explicit descriptions of which are precisely what is expected of a process calculus. If $M(0)$ never interacts, the execution of $M(0)$ in a higher order programming language would result in a loop, a computational behaviour that is quite different from that of the command *skip*. However in the standard semantics of the value-passing calculi $M(0)$ is strongly bisimilar to $\mathbf{0}$. In a basic model processes like (1) should be banned.

The above discussions lead to the following design principle. A value-passing calculus consists of a first order theory and a labeled transition system. The former provides both the value terms and the boolean expressions. The latter defines the semantics of the value-passing processes. The calculus is designed by taking the following into consideration.

1. To make sure that the first order theory provides a right support to the operational semantics, the theory is supposed to be complete for the set of the quantifier free theorems.
2. To guarantee that the first order theory does not interfere with the computations/interactions defined by the labeled transition system, the formulas admissible in a value-passing calculus should only contain constructors that generate the universe of values; they should not contain any functions that compute on the elements of the universe.
3. To keep the value-passing calculi at the right level of abstraction, it would be better not to define recursions by recursive definitions parameterized over value variables. The replication operator is sufficient.

The aim of this paper is to develop a rigid theory of the value-passing calculi designed with the above remarks in mind. The theory is general enough so that it can be readily applied to any particular value-passing calculus. It is also formal enough so that many questions about the value-passing calculi can be addressed.

The paper is structured as follows. Section 2 reviews the relevant terminologies in mathematical logic. Section 3 studies the operational and the observational semantics of the value-passing calculi. Section 4 takes a look at symbolic approximation to the absolute equality. Section 5 provides a proof system for the finite terms. Section 6 discusses the expressiveness requirement for the value-passing calculi. Section 7 applies the methodology to the value-passing calculus $\mathbb{VPC}$ defined over the Peano Arithmetics. Section 8 concludes with discussions on future research.

BA	$P\{P_1/X_1, \ldots, P_n/X_n\}$	P is a tautology
EQ1	$t = t$	
EQ2	$s = t \Rightarrow t = s$	
EQ3	$r = s \wedge s = t \Rightarrow r = t$	
CG1	$\bigwedge_{i=1}^{k} t_i = t'_i \Rightarrow \mathsf{f}(t_1, \ldots, t_k) = \mathsf{f}(t'_1, \ldots, t'_k)$	f is a k-ary function
CG2	$\bigwedge_{i=1}^{k} t_i = t'_i \Rightarrow \mathsf{r}(t_1, \ldots, t_k) \Rightarrow \mathsf{r}(t'_1, \ldots, t'_k)$	r is a k-ary relation
FO1	$\forall x.\phi \Rightarrow \phi\{t/x\}$	
FO2	$\phi \Rightarrow \forall x.\phi$	x not in ϕ
FO3	$(\forall x.(\varphi \Rightarrow \psi)) \Rightarrow (\forall x.\varphi \Rightarrow \forall x.\psi)$	

Fig. 1. Logical Axioms of Σ

2 Decidable Theory

Let $\mathbf{N}$ be the set of natural numbers. A *vocabulary* $\Sigma = (\mathsf{F}, \mathsf{R}, \mathsf{a})$ consists of two disjoint nonempty countable sets and one function: F is a finite set of *function symbols*; R is a finite set of *relation symbols*; and $\mathsf{a} : \mathsf{F} \cup \mathsf{R} \to \mathbf{N}$ is an *arity function* that maps an element of F onto a natural number and an element of R onto a nonzero natural number. A symbol in $\mathsf{F} \cup \mathsf{R}$ is *k-ary* if it is mapped onto k under a. A *constant* is a 0-ary function symbol. It is always assumed that F contains at least one constant and that R contains the *equality relation* $=$.

For each vocabulary Σ there is a countable set $\mathsf{V}_\Sigma = \{x, y, z, \ldots\}$ of *Σ-variables*. The set T_Σ of *Σ-terms*, ranged over by r, s, t, is defined as follows:

- $\mathsf{V}_\Sigma \subseteq \mathsf{T}_\Sigma$.
- If f is a k-ary function symbol and $t_1, \ldots, t_k$ are Σ-terms, then $\mathsf{f}(t_1, \ldots, t_k)$ is a Σ-term.

A Σ-term is *closed* if it does not contain any Σ-variable, it is *open* otherwise. The set of closed Σ-terms is denoted by T^0_Σ.

The set E_Σ of *Σ-expressions*, ranged over by ϕ, φ, ψ, is defined as follows:

- The logical *false* $\bot$ is a Σ-expression.
- If r is a k-ary relation symbol and $t_1, \ldots, t_k$ are Σ-terms, then $\mathsf{r}(t_1, \ldots, t_k)$ is an *atomic* Σ-expression.
- If φ, ψ are Σ-expressions, then $\varphi \Rightarrow \psi$ is a Σ-expression.
- If ϕ is a Σ-expression and x is a Σ-variable, then $\forall x.\phi$ is a Σ-expression, where $\forall$ is the *universal quantifier*.

The Σ-variable x in $\forall x.\phi$ is *bound*. A Σ-variable is *free* if it is not bound. A *Σ-sentence* is a Σ-expression that does not contain any free Σ-variables. The set of Σ-sentences is denoted by E^0_Σ. A *boolean* Σ-expression is a quantifier free Σ-expression. In sequel we shall freely use the derived logical connectives $\top, \neg, \wedge, \vee, \Leftrightarrow, \exists$.

The *first order logic over Σ* is the recursive set of the *first order logical axioms* defined in Fig. 1. The axiom schema BA actually stands for a recursive set of *boolean axioms*, each obtained from a boolean tautology by instantiating all the propositional variables. The EQ-axioms are about the equivalence property, and

PA1	$\forall x.(\mathsf{s}(x) \neq 0)$
PA2	$\forall xy.(\mathsf{s}(x) = \mathsf{s}(y) \Rightarrow x = y)$
PA3	$\forall x.(x = 0 \vee \exists y.\mathsf{s}(y) = x)$
PA4	$\forall x.(x < \mathsf{s}(x))$
PA5	$\forall xy.(x < y \Rightarrow \mathsf{s}(x) \leq y)$
PA6	$\forall xy.(\neg(x < y) \Leftrightarrow y \leq x)$
PA7	$\forall xy.((x < y) \wedge (y < z) \Rightarrow x < z)$

Fig. 2. First Order Theory PA

the CG-axioms formalize the congruence property. The FO-axioms state the provability of the universally quantified Σ-expressions. In both BA and FO1 the meta operation substitution is used.

Given a recursive set Γ of Σ-expressions, a *proof* of ψ from Γ is a finite sequence $(\phi_1, \ldots, \phi_n)$ of Σ-expressions such that ϕ_n is ψ and one of the following properties holds for each $i \leq n$:

- ϕ_i is a logical axiom;
- $\phi_i \in \Gamma$;
- There are two Σ-expressions φ and $\varphi \Rightarrow \phi_i$ in the proof $(\phi_1, \ldots, \phi_{i-1})$.

A Σ-expression ψ is a *Γ-theorem*, notation $\Gamma \vdash \psi$, if there is a proof of ψ from Γ, and it is a *theorem*, notation $\vdash \psi$, if Γ is the empty set. A set Γ of Σ-expressions is *inconsistent* if $\Gamma \vdash \bot$; it is *consistent* otherwise. We sometimes write $s =_\Gamma t$ for $\Gamma \vdash s = t$.

A *first order theory over Σ* is a consistent recursive set Th of Σ-sentences, the elements of Th are called *nonlogical axioms*.

In the context of present paper the most useful first order theory is Presburger Arithmatic [Pre29]. This is what one gets if the multiplication operator '$\times$' is removed from the Peano Arithmetic. For the reasons explained in Section 1 we shall also leave out the addition operator '+'. The axioms of our theory PA are given in Fig. 2, in which $x \neq y$ stands for $\neg(x = y)$ and $x \leq y$ for $x = y \vee x < y$. For a natural number i, let $\underline{i}$ denote the *numeral*

$$\underbrace{\mathsf{s}(\mathsf{s}(\ldots \mathsf{s}}_{i \text{ times}}(0)\ldots)).$$

Similarly we write $\mathsf{s}^i(x)$ for the open Σ_{PA}-term

$$\underbrace{\mathsf{s}(\mathsf{s}(\ldots \mathsf{s}}_{i \text{ times}}(x)\ldots)).$$

Presburger [Pre29] proved that remarkably his arithmetic, and consequently the theory PA defined in Fig. 2, is decidable. This is the foundation for the value-passing calculi studied in the rest of the paper.

Theorem 1. *PA is decidable.*

3 Value-Passing Calculus

According to our discussions in Section 1, we shall focus on the value-passing calculi defined in terms of decidable first order theories. Throughout this paper we assume that Th is a decidable first order theory of type Σ. The value-passing calculus defined on top of Th is denoted by $\mathbb{VPC}_{\mathsf{Th}}$. If Th is PA, the subscript in $\mathbb{VPC}_{\mathsf{Th}}$ is omitted. The abbreviation will be justified in Section 7.

All process calculi are defined in terms of *names*. The set $\mathcal{N}$ of names is ranged over by a, b, c, d, e, f, g, h. The set $\overline{\mathcal{N}}$ of conames is $\{\overline{a} \mid a \in \mathcal{N}\}$. A *substitution* is a partial map $\sigma : \mathsf{V}_\Sigma \rightharpoonup \mathsf{T}_\Sigma$ whose domain of definition is finite. An *assignment* is a partial map $\rho : \mathsf{V}_\Sigma \rightharpoonup \mathsf{T}^0_\Sigma$ whose domain of definition is cofinite. A substitution is often denoted explicitly by $\{t_1/x_1, \ldots, t_n/x_n\}$. The notations $\rho[x \leftarrow t]$ and $\sigma[x \leftarrow t]$ are understood in the standard interpretation.

The set $\mathcal{T}_{\mathbb{VPC}_{\mathsf{Th}}}$ of the $\mathbb{VPC}_{\mathsf{Th}}$-*terms*, ranged over by R, S, T and their decorated forms, is defined by the following BNF:

$$T := \sum_{i\in I} \varphi_i a(x).T_i \mid \sum_{i\in I} \varphi_i \overline{a}(t_i).T_i \mid T \mid T' \mid (c)T \mid \varphi T \mid\, !a(x).T \mid\, !\overline{a}(t).T,$$

where φ_i is a boolean Σ-expression, and I is a finite indexing set. The notation $\sum_{i\in\{1,\ldots,n\}} \varphi_i\lambda_i.T_i$ stands for either $\sum_{i\in I} \varphi_i a(x).T_i$ or $\sum_{i\in I} \varphi_i \overline{a}(t_i).T_i$. The prefix $a(x)$ is an input primitive that binds the Σ-variable (henceforth just variable) x, and the prefix $\overline{a}(t)$ is an output primitive. We write $fv(\_)$, respectively $bv(\_)$, for the function that returns the set of the free variables, respectively the bound variables; and let $v(\_)$ be $fv(\_) \cup bv(\_)$. A $\mathbb{VPC}_{\mathsf{Th}}$-term is *closed* if it does not contain any free variables. Otherwise it is called an *open* $\mathbb{VPC}_{\mathsf{Th}}$-term. A closed $\mathbb{VPC}_{\mathsf{Th}}$-term is also called a $\mathbb{VPC}_{\mathsf{Th}}$-*process*. We write $\mathcal{P}_{\mathbb{VPC}_{\mathsf{Th}}}$ for the set of the $\mathbb{VPC}_{\mathsf{Th}}$-processes, ranged over by L, M, N, O, P, Q. For clarity we shall write $A(x, y) \stackrel{\text{def}}{=} T$ for instance to indicate that $A(x, y)$ is a shorthand for T with x, y as the only free variables. The notation $A(s, t)$ denotes $T\{s/x, t/y\}$. The *composition* $T \mid T'$ and the *localization* $(c)T$ are standard constructions. We write $\prod_{1\le i\le n} T_i$ for the composition $T_1 \mid T_2 \mid \ldots \mid T_n$. The $\mathbb{VPC}_{\mathsf{Th}}$-term $\sum_{i\in I} \varphi_i\lambda_i.T_i$ is a *conditional guarded choice*, and for each $i \in I$ the component $\varphi_i\lambda_i.T_i$ is a *summand*. The condition φ_i is often omitted if it is $\top$. There is essentially a unique guarded choice, noted $\mathbf{0}$, whose index set is the empty set. Often we write $\varphi_1\lambda_1.T_1 + \ldots + \varphi_n\lambda_n.T_n$ for $\sum_{i\in\{1,\ldots,n\}} \varphi_i\lambda_i.T_i$. It should be remarked that in $\sum_{i\in I} \varphi_i\lambda_i.T_i$ the constructor is $\sum_{i\in I} \varphi_i\lambda_i.\_$. The *conditional* φT is often written as *if* φ *then* T. The two leg conditional *if* φ *then* S *else* T can be defined by $\varphi S \mid \neg\varphi T$. The $\mathbb{VPC}_{\mathsf{Th}}$-term $!\nu.T$ is a *guarded replication*. We shall freely use the guarded fixpoint terms of the form $\mu X.E$ where X is a process variable whose occurrences in E are all under some prefixes. The fixpoint construction $\mu X.E$ can be encoded by $(c)(E\{c(z).\mathbf{0}/X\} \mid !\overline{c}(r).E\{c(z).\mathbf{0}/X\})$, where c is fresh and r is a closed term. So the guarded fixpoint operator does not introduce extra expressive power [FL10]. A $\mathbb{VPC}_{\mathsf{Th}}$-term is *finite* if it does not contain any occurrences of the replication operator; it is a *finite control* term if it contains only the conditional guarded choice operator and the fixpoint operator.

Action

$$\frac{}{\sum_{i\in I}\varphi_i a(x).T_i \xrightarrow{a(t)} T_i\{t/x\}} \; \begin{array}{l} i\in I,\\ t\in \mathsf{T}^0_\Sigma,\\ \mathsf{Th}\vdash\varphi_i. \end{array} \qquad \frac{}{\sum_{i\in I}\varphi_i \overline{a}(t_i).T_i \xrightarrow{\overline{a}(t_i)} T_i} \; \begin{array}{l} i\in I,\\ t_i\in \mathsf{T}^0_\Sigma,\\ \mathsf{Th}\vdash\varphi_i. \end{array}$$

Composition

$$\frac{S \xrightarrow{\lambda} S'}{S\,|\,T \xrightarrow{\lambda} S'\,|\,T} \qquad \frac{S \xrightarrow{a(t)} S' \quad T \xrightarrow{\overline{a}(t)} T'}{S\,|\,T \xrightarrow{\tau} S'\,|\,T'}$$

Localization

$$\frac{T \xrightarrow{\lambda} T'}{(c)T \xrightarrow{\lambda} (c)T'}\ c \text{ is not in } \lambda.$$

Condition

$$\frac{T \xrightarrow{\lambda} T'}{\varphi T \xrightarrow{\lambda} T'}\ \mathsf{Th}\vdash\varphi.$$

Recursion

$$\frac{}{!a(x).T \xrightarrow{a(t)} T\{t/x\}\,|\,!a(x).T}\ t\in\mathsf{T}^0_\Sigma. \qquad \frac{}{!\overline{a}(t).T \xrightarrow{\overline{a}(t)} T\,|\,!\overline{a}(t).T}\ t\in\mathsf{T}^0_\Sigma.$$

Fig. 3. Concrete Semantics

3.1 Concrete Semantics

In this section we define the so-called *concrete semantics*, which is given by the labeled transition system in Fig. 3, where the symmetric versions of two composition rules have been omitted. Although the semantics is defined for all $\mathbb{VPC}_{\mathsf{Th}}$-terms, only the behaviors of the $\mathbb{VPC}_{\mathsf{Th}}$-processes are completely characterized. If Th is the Peano Arithmetic PA, then under our semantics the $\mathbb{VPC}_{\mathsf{Th}}$-term *if* $\underline{0} \le x$ *then* $\overline{a}(\underline{0})$ can perform an action, but the obviously equivalent $\mathbb{VPC}_{\mathsf{Th}}$-term *if* $x = \underline{0}$ *then* $\overline{a}(\underline{0})$ *else* $\overline{a}(\underline{0})$ cannot do anything.

The reader must have noticed that the rule for the output prefix in our concrete semantics appears different from the standard treatment. In the value-passing calculi defined in terms of a model [Mil89a], the term in an output prefix must be calculated to a value, an element of the universe, before it is exported. In our approach however the Σ-term is exported as it is. There are two reasons. One is that at our abstract model, there is no way to talk about calculation of terms. But the much more important reason, alluded in Section 1, is that the functional power of the oracle is undesirable. The first order theory provides a universe of values, whereas the value-passing calculus does the calculation. If we maintain a separation between the Σ-terms and the calculations of the Σ-terms, there is no need to calculate any closed Σ-terms since every closed Σ-term is already a 'value'.

Let us see two examples. Let A be $(a)(\overline{a}(\underline{0}) \mid \mu X.a(x).(\overline{a}(\mathsf{s}(x)) \mid (\tau.X + \overline{b}(x))))$. A typical action sequence of A is $A\underbrace{\xrightarrow{\tau}\xrightarrow{\tau}\ldots\xrightarrow{\tau}\xrightarrow{\tau}}_{2n+1 \text{ times}}\xrightarrow{\overline{b}(\underline{n})}\mathbf{0}$, where $n \geq 0$. As this example shows, the calculation of the numeral is explicitly demonstrated. The second example is about the encoding of the minimization operator. It is given by $(a)(\overline{a}(\underline{0}) \mid \mu X.a(x).(\overline{a}(\mathsf{s}(x)) \mid \mathit{if}\ \varphi\ \mathit{then}\ \overline{b}(x)\ \mathit{else}\ \tau.X))$. Notice that if no numerals satisfy φ then the process diverges.

We write $\Longrightarrow$ for the reflexive and transitive closure of $\xrightarrow{\tau}$, and $\overset{\lambda}{\Longrightarrow}$ for the composition $\Longrightarrow\xrightarrow{\lambda}\Longrightarrow$.

3.2 Absolute Equality

A first attempt to define the bisimulations for $\mathbb{VPC}_{\mathsf{Th}}$ is to reiterate the definition from the theory of CCS for all $\mathbb{VPC}_{\mathsf{Th}}$-terms. This would require that $S \xrightarrow{\lambda} S'$ should be bisimulated by $T \overset{\widehat{\lambda}}{\Longrightarrow} T'$ whenever S is bisimilar to T. Consider however the $\mathbb{VPC}$-terms $\overline{a}(\underline{0}).S$ and $\mathit{if}\ x = \underline{0}\ \mathit{then}\ \overline{a}(\underline{0}).S\ \mathit{else}\ \overline{a}(\underline{0}).S$. Intuitively these two $\mathbb{VPC}$-terms are equivalent. But the action $\overline{a}(\underline{0}).S \xrightarrow{\overline{a}(\underline{0})} S$ cannot be simulated by any action of $\mathit{if}\ x = \underline{0}\ \mathit{then}\ \overline{a}(\underline{0}).S\ \mathit{else}\ \overline{a}(\underline{0}).S$. There are two ways to bypass the problem. One is to confine our attention to processes. This would be a reasonable choice if the operational semantics is formulated in a concrete manner. The other is to apply a symbolic approach, which would of course fit very well with the symbolic operational semantics. Before we take a look at these two solutions, we shall apply to $\mathbb{VPC}_{\mathsf{Th}}$ the model independent approach developed in [Fu12b]. The equality so obtained provides not only the intuition, but also a standard to compare against.

Process equivalences are observational. A process is observable if it may interact with another process.

Definition 1. *A process P is* observable*, notation $P{\Downarrow}$, if $\exists \lambda, P'.\ P \Longrightarrow \xrightarrow{\lambda} P'$.*

Now whatever an observational equivalence is, it must not identify an observable process with an unobservable process. Hence the next definition.

Definition 2. *A binary relation $\mathcal{R}$ is* equipollent *if $P{\Downarrow} \Leftrightarrow Q{\Downarrow}$ whenever $P\mathcal{R}Q$.*

Now suppose two processes P, Q are observationally equivalent. A third process, say O, cannot detect any difference between P, Q by interacting with them. If we think of it, the fact that O cannot tell P, Q apart is best interpreted as saying that $P \mid O$ and $Q \mid O$ are observationally equivalent. Now trivially, P and Q cannot be distinguished by any process that does not interact at a particular channel name, say c. If one looks at the same thing from another angle, one easily sees that $(c)P$ must be observationally equivalent to $(c)Q$ as well.

Definition 3. *A relation $\mathcal{R}$ is* extensional *if the following hold: (i) If $L\mathcal{R}M$ and $P\mathcal{R}Q$ then $(L \mid P)\ \mathcal{R}\ (M \mid Q)$. (ii) If $P\mathcal{R}Q$ then $(c)P\ \mathcal{R}\ (c)Q$ for all $c \in \mathcal{N}$.*

If two processes are equivalent, they should be able to maintain the equivalence after one thousand years. The minimal condition making sure that this can be achieved is the bisimulation property of Milner [Mil89a] and Park [Par81]. Following the idea of Fu [Fu12b], we actually will use a stronger version of the bisimulation introduced by van Glabbeek and Weijland [vGW89]. In the following definition, the notation $\mathcal{R}^{-1}$ stands for the inverse of $\mathcal{R}$.

Definition 4. *A binary relation is a* bisimulation *if the following hold:*

1. *If* $Q\mathcal{R}^{-1}P \xrightarrow{\tau} P'$ *then one of the following statements is valid.*
   (a) $Q \Longrightarrow Q'\mathcal{R}^{-1}P'$ *and* $Q'\mathcal{R}^{-1}P$ *for some* Q'.
   (b) $Q \Longrightarrow Q''\mathcal{R}^{-1}P$ *for some* Q'' *such that* $Q'' \xrightarrow{\tau} Q'\mathcal{R}^{-1}P'$ *for some* Q'.
2. *If* $P\mathcal{R}Q \xrightarrow{\tau} Q'$ *then one of the following statements is valid.*
   (a) $P \Longrightarrow P'\mathcal{R}Q'$ *and* $P'\mathcal{R}Q$ *for some* P'.
   (b) $P \Longrightarrow P''\mathcal{R}Q$ *for some* P'' *such that* $P'' \xrightarrow{\tau} P'\mathcal{R}Q'$ *for some* P'.

A basic assumption in the theory of computation is that a divergent computation is different from a computation that terminates. Often time the probability for a real program to diverge is zero. For such a program, divergence is a potential, not an inevitability. A condition that takes into account of this potentiality while upholding the bisimulation property is what we call codivergence requirement. It was first proposed by Priese [Pri78].

Definition 5. *A relation* $\mathcal{R}$ *is* codivergent *if the following statements are valid:*

- *If* $P\mathcal{R}Q \xrightarrow{\tau} Q_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} Q_n \xrightarrow{\tau} \cdots$ *is an infinite internal action sequence, then there must be some* $k \geq 1$ *and* P' *such that* $P \stackrel{\tau}{\Longrightarrow} P'\ \mathcal{R}\ Q_k$.
- *If* $Q\mathcal{R}^{-1}P \xrightarrow{\tau} P_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} P_n \xrightarrow{\tau} \cdots$ *is an infinite internal action sequence, then there must be some* $k \geq 1$ *and* Q' *such that* $Q \stackrel{\tau}{\Longrightarrow} Q'\ \mathcal{R}\ P_k$.

We have introduced four conditions for the equivalences on evolving processes, which are minimal from the point of view of interaction and computation. We turn these minimal conditions into defining properties of process equality.

Definition 6. *The* absolute equality $=_{Th}$ *is the largest reflexive, equipollent, extensional, codivergent bisimulation on* $\mathcal{P}_{\mathbb{VPC}_{Th}}$.

The well-definedness of Definition 6 is due to the fact that its defining properties are stable under set unions. The definition is completely model independent as long as we only consider those models that have the composition and localization operators and enjoy a dichotomy between the internal actions and the external interactions. From the point of view of equality reasoning, the absolute equality is too abstract. It would be very helpful to work out an external characterization of the absolute equality. This is what we are going to do next for $\mathbb{VPC}_{\mathsf{Th}}$. Before that, we state a useful lemma about computation, the Bisimulation Lemma [Fu12b]. The property stated in the lemma is called X-property by De Nicola, Montanari and Vaandrager [DNMV90].

Lemma 1. *If* $P \Longrightarrow P' =_{Th} Q$ *and* $Q \Longrightarrow Q' =_{Th} P$, *then* $P =_{Th} Q$.

Once the equality relation has been defined, a formal classification of the internal actions can be given. We say that S evolves to T in a *computation step*, notation $S \to T$, if $S \xrightarrow{\tau} T$ and $S =_{\mathsf{Th}} T$, and that S evolves to T in a *change-of-state* internal action, notation $S \xrightarrow{\iota} T$, if $S \xrightarrow{\tau} T$ and $S \neq_{\mathsf{Th}} T$. The reflexive and transitive closure of $\to$ will be denoted by $\to^*$.

The bisimulation property can now be defined in a more informative way. If $P =_{\mathsf{Th}} Q \xrightarrow{\tau} Q'$, then the simulation by P could be vacuous if $Q =_{\mathsf{Th}} Q'$; otherwise it must take the form $P \to^* P'' \xrightarrow{\iota} P'$ such that $P'' =_{\mathsf{Th}} Q$ and $P' =_{\mathsf{Th}} Q'$. Similarly if $P =_{\mathsf{Th}} Q \xrightarrow{\overline{a}(t)} Q'$, then the simulation of the output action must take the form $P \to^* P'' \xrightarrow{\overline{a}(t)} P'$ such that $P'' =_{\mathsf{Th}} Q$ and $P' =_{\mathsf{Th}} Q'$. This is the external bisimulation property we shall define in the next section.

3.3 External Bisimulation

External bisimulations are meant to give an alternative characterization of the absolute equality in terms of explicit simulation of *every* action. In addition to the property of Definition 4, external bisimulations must explain how the external actions are bisimulated.

Definition 7. *A codivergent bisimulation $\mathcal{R}$ on $\mathcal{P}_{\mathbb{VPC}_{Th}}$ is a $\mathbb{VPC}_{Th}$-*bisimulation *if the following statements are valid for every $\lambda \neq \tau$.*

1. *If $Q\mathcal{R}^{-1}P \xrightarrow{\lambda} P'$ then $Q \Longrightarrow Q'' \xrightarrow{\lambda} Q'\mathcal{R}^{-1}P'$ and $P\mathcal{R}Q''$ for some Q', Q''.*
2. *If $P\mathcal{R}Q \xrightarrow{\lambda} Q'$ then $P \Longrightarrow P'' \xrightarrow{\lambda} P'\mathcal{R}Q'$ and $P''\mathcal{R}Q$ for some P', P''.*

*The $\mathbb{VPC}_{Th}$-*bisimilarity *$\simeq_{Th}$ is the largest $\mathbb{VPC}_{Th}$-bisimulation.*

By constructing the relation inductively from $\simeq_{\mathsf{Th}}$ that closes up under composition and localization, one can easily prove the following lemma.

Lemma 2. *The external bisimilarity $\simeq_{Th}$ is extensional.*

The above lemma and the Bisimulation Lemma is the only thing we need to establish Proposition 1, which proves the correctness of Definition 7.

Proposition 1. *The relation $\simeq_{Th}$ coincides with the absolute equality $=_{Th}$.*

Proof. The inclusion $\simeq_{\mathsf{Th}} \subseteq =_{\mathsf{Th}}$ is immediate from Lemma 2. The proof of the reverse inclusion is standard using Bisimulation Lemma. Processes of the form $a(x).\mathit{if}\ x = \_\ \mathit{then}\ \overline{c}(\_)\ \mathit{else}\ \overline{d}(\_)$ and of the form $\overline{a}(\_) + \overline{a}(\_).\overline{c}(\_)$, with the names c, d chosen properly, are crucial to deriving the external bisimulation property. A detailed proof of a similar result in π-calculus is given in [FZ11]. □

Both the absolute equality and the external bisimilarity are relations on the processes. They can be extended to the $\mathbb{VPC}_{\mathsf{Th}}$-terms in the standard manner.

Definition 8. *$S \simeq_{Th} T$ if and only if $S\rho \simeq_{Th} T\rho$ for every assignment ρ whose domain of definition is disjoint from $bv(S \mid T)$.*

A standard argument suffices to show that the relation $\simeq_{\mathsf{Th}}$, and consequently the relation $=_{\mathsf{Th}}$ as well, is closed under all the process operations.

Proposition 2. *The absolute equality is equivalent and congruent.*

Action

$$\frac{}{\sum_{i\in I}\varphi_i\lambda_i.T_i \xrightarrow{\lambda_i}_{\varphi_i} T_i} \quad i \in I.$$

Composition

$$\frac{S \xrightarrow{\lambda}_{\varphi} S'}{S \,|\, T \xrightarrow{\lambda}_{\varphi} S' \,|\, T} \qquad \frac{S \xrightarrow{a(x)}_{\varphi} S' \quad T \xrightarrow{\overline{a}(t)}_{\psi} T'}{S \,|\, T \xrightarrow{\tau}_{\varphi\psi} S'\{t/x\} \,|\, T'}$$

Localization

$$\frac{T \xrightarrow{\lambda}_{\varphi} T'}{(c)T \xrightarrow{\lambda}_{\varphi} (c)T'} \quad c \text{ is not in } \lambda.$$

Condition

$$\frac{T \xrightarrow{\lambda}_{\varphi} T'}{\phi T \xrightarrow{\lambda}_{\phi\varphi} T'}$$

Recursion

$$\frac{}{!a(x).T \xrightarrow{a(x)}_{\top} T \,|\, !a(x).T} \qquad \frac{}{!\overline{a}(t).T \xrightarrow{\overline{a}(t)}_{\top} T \,|\, !\overline{a}(t).T}$$

Fig. 4. Symbolic Semantics

4 Symbolic Semantics

The absolute equality is not very convenient. Hennessy and Lin [HL95] address the issue by introducing symbolic bisimulations. The advantage of the symbolic approach is that it allows one to make full use of the decidable fragments of the logics of the value-passing calculi when constructing equivalence checking algorithms. In [HL95] the symbolic bisimilarity is defined as general as possible so that a coincidence result could be achieved. We shall be less ambitious in this paper. Symbolic bisimilarity is in our opinion a decidable approximation of the absolute equality. Our motivation for the symbolic bisimilarity is two folds: (i) the symbolic bisimilarity should be sound, meaning that it should be a subset of the absolute equality; (ii) ideally the symbolic bisimilarity is complete on the decidable subsets of the absolute equality. The latter is much more difficult to come by than the former.

To define the symbolic bisimulations, we need to introduce the symbolic operational semantics. This would not be a big issue had we dropped the conditionals. So the key is to define for instance the semantics of the $\mathbb{VPC}_{\mathsf{Th}}$-term *if* φ *then* $\overline{a}(\underline{0})$ *else* $\overline{b}(\underline{0})$ where the boolean Σ-expression φ contains free variables. The *symbolic semantics* [HL95, HL96] solves the problem by introducing conditional actions. The syntax of a transition in the symbolic semantics is a tuple of the form $T \xrightarrow{\lambda}_{\varphi} T'$, formalizing the idea that T may perform the action λ

under the condition that the boolean Σ-expression φ is a Th-theorem. The set of the action labels for the symbolic semantics is

$$\{a(x), \overline{a}(t) \mid a \in \mathcal{N}, x \in \mathsf{V}_\Sigma, t \in \mathsf{T}_\Sigma\} \cup \{\tau\},$$

also ranged over by λ. The labeled transition system is defined in Fig. 4. Again the symmetric versions of the composition rules have been omitted. The treatment of the input prefix is in a late style, which is better suited for algorithmic investigations. The symbolic semantics is stable under substitution in the sense of the following lemma.

Lemma 3. *If $S \xrightarrow{\lambda}_\varphi T$ then $S\sigma \xrightarrow{\lambda\sigma}_{\varphi\sigma} T\sigma$ for every substitution σ. On the other hand, if $S\sigma \xrightarrow{\lambda'}_{\varphi'} T'$ for some substitution σ, then there must exist some λ, φ, T such that $\lambda' = \lambda\sigma$, $\varphi' = \varphi\sigma$, $T' \equiv T\sigma$ and $S \xrightarrow{\lambda}_\varphi T$.*

We will abbreviate $\xrightarrow{\tau}_{\varphi_1} \ldots \xrightarrow{\tau}_{\varphi_n}$ to $\Longrightarrow_{\varphi_1 \ldots \varphi_n}$, and $\Longrightarrow_\varphi \xrightarrow{\lambda}_{\varphi'} \Longrightarrow_{\varphi''}$ to $\overset{\lambda}{\Longrightarrow}_{\varphi\varphi'\varphi''}$. We remark that $T \Longrightarrow_\top T$.

It's time to look at some examples. The following three terms are in $\mathbb{VPC}$:

$$\begin{aligned}
L(y) &\overset{\text{def}}{=} \textit{if } \underline{0} \leq y \textit{ then } \overline{b}(\underline{0}),\\
M(y) &\overset{\text{def}}{=} (a)(\overline{a}(\underline{0}) \mid \mu X.a(x).(\overline{a}(\mathsf{s}(x)) \mid (\tau.X + \textit{if } y = x \textit{ then } \overline{b}(\underline{0})))),\\
N(y) &\overset{\text{def}}{=} (a)(\overline{a}(\underline{0}) \mid \mu X.a(x).(\overline{a}(\mathsf{s}(x)) \mid (\tau.X + \textit{if } y = x \textit{ then } \overline{b}(y)))).
\end{aligned}$$

The process $L(y)$ may perform only one action $L(y) \xrightarrow{\overline{b}(\underline{0})}_{\underline{0} \leq y} \mathbf{0}$. The process $M(y)$ has an interesting action sequence $M(y) \Longrightarrow_\top \xrightarrow{\overline{b}(\underline{0})}_{y=\underline{n}} \mathbf{0}$. Similarly $N(y)$ has a similar action sequence $N(y) \Longrightarrow_\top \xrightarrow{\overline{b}(y)}_{y=\underline{n}} \mathbf{0}$. In the second and third examples the set of possible conditions is $\{y{=}\underline{0}, y{=}\underline{1}, y{=}\underline{2}, \ldots, y{=}\underline{n}, \ldots\}$. What the second example tells us is that even with a finite description and a finite set of potential external actions, the set of the conditions that enable an action of a $\mathbb{VPC}$-term, like $\overline{b}(\underline{0})$, could be infinite.

The symbolic semantics is a correct extension of the concrete semantics. This is established in the next two lemmas whose proofs are simple induction on derivation.

Lemma 4. *The symbolic semantics is sound with respect to the concrete semantics in the following sense:*
(i) If $S \xrightarrow{\tau}_\varphi T$ for some Th-theorem φ then $S \xrightarrow{\tau} T$.
(ii) If $S \xrightarrow{\overline{a}(t)}_\varphi T$ for some Th-theorem φ and some $t \in \mathsf{T}^0_\Sigma$ then $S \xrightarrow{\overline{a}(t)} T$.
(iii) If $S \xrightarrow{a(x)}_\varphi T$ for some Th-theorem φ then $S \xrightarrow{a(t)} T\{t/x\}$ for every $t \in \mathsf{T}^0_\Sigma$.

Lemma 5. *The symbolic semantics is complete with respect to the concrete semantics in the following sense:*
(i) If $S \xrightarrow{\tau} T$ then $S \xrightarrow{\tau}_\varphi T$ for some Th-theorem φ.

(ii) If $S \xrightarrow{\overline{a}(t)} T$ *then* $S \xrightarrow{\overline{a}(t)}_{\varphi} T$ *for some* Th*-theorem* φ.
(iii) If $S \xrightarrow{a(t)} T$ *then* $S \xrightarrow{a(x)}_{\varphi} T'$ *for some* Th*-theorem* φ *and some* x, T' *such that* $T \equiv T'\{t/x\}$.

4.1 Symbolic Bisimulation

In the symbolic approach, is it reasonable to require that $S \xrightarrow{\tau}_{\varphi} S'$ be simulated by $T \Longrightarrow_{\varphi'} T'$ for bisimilar S and T such that $\mathsf{Th} \vdash \varphi \Rightarrow \varphi'$? Again let's take a look at the $\mathbb{VPC}$-terms $\varphi\tau.S$ and *if* $x = \underline{0}$ *then* $\varphi\tau.S$ *else* $\varphi\tau.S$, where $x \notin fv(\varphi)$. The two $\mathbb{VPC}$-terms are equivalent by all reasonable criteria. But $\varphi\tau.S \xrightarrow{\tau}_{\varphi} S$ cannot be simulated by *if* $x = \underline{0}$ *then* $\varphi\tau.S$ *else* $\varphi\tau.S$ in the above required manner since in the latter term the τ-action can only be fired under a condition strictly stronger than φ. But notice that $(x = \underline{0}) \wedge \varphi \vee (x \neq \underline{0}) \wedge \varphi \Leftrightarrow \varphi$ is a PA-theorem. Under either $(x = \underline{0}) \wedge \varphi$ or $(x \neq \underline{0}) \wedge \varphi$ the $\mathbb{VPC}$-term *if* $x = \underline{0}$ *then* $\varphi\tau.S$ *else* $\varphi\tau.S$ may evolve into S. This leads to the idea of finding a collection of boolean expressions such that the disjunction of the collection is weaker than φ. If under each of the conditions the simulation can be done, then there is a simulation. A first attempt to define a symbolic counterpart of the Milner-Park bisimilarity could be as follows:

> A symmetric relation $\mathcal{R}$ on $\mathcal{T}_{\mathbb{VPC}_{\mathsf{Th}}}$ is a ?-bisimulation if the following condition is met whenever $S\mathcal{R}T$:
>
> If $S \xrightarrow{\lambda}_{\varphi} S'$ then there is a class $\{T \overset{\widehat{\lambda_i}}{\Longrightarrow}_{\varphi_i} T'_i\}_{i \in I}$ such that $\mathsf{Th} \vdash \varphi\varphi_i \Rightarrow \lambda = \lambda_i$ and $(\varphi\varphi_i S', \varphi\varphi_i T'_i) \in \mathcal{R}$ for every $i \in I$.
>
> Let $\simeq^?$ be the largest ?-bisimulation.

In the above definition, $\lambda = \lambda_i$ stands for $\top$ if λ, λ_i are syntactically the same; it is $t = t'$ if $\lambda \equiv \overline{a}(t)$ and $\lambda_i \equiv \overline{a}(t')$ for some name a; otherwise $\lambda = \lambda_i$ stands for $\bot$. The relation $\simeq^?$ is not very useful. Consider the $\mathbb{VPC}$-processes $R(y), S(y), T(y)$ defined as follows:

$$
\begin{aligned}
R(y) &\overset{\text{def}}{=} (b)(\overline{b}(y) \mid \mu X.b(z).\overline{r}(z).\mathit{if}\ \underline{0} < z\ \mathit{then}\ (\overline{b}(\mathsf{p}(z)) \mid X)),\\
S(y) &\overset{\text{def}}{=} \tau.T(y) + \mathit{if}\ \underline{0} \leq y\ \mathit{then}\ \tau.R(y),\\
T(y) &\overset{\text{def}}{=} (a)(\overline{a}(\underline{0}) \mid \mu X.a(x).(\overline{a}(\mathsf{s}(x))\\
&\qquad \mid (\tau.X + \mathit{if}\ \underline{0} \leq y \leq x\ \mathit{then}\ \tau.(\mathit{if}\ y = x\ \mathit{then}\ \tau.R(y)\ \mathit{else}\ \overline{e}(\underline{0}))))).
\end{aligned}
$$

In the definition of $R(y)$ the term $\mathsf{p}(z)$ is the predecessor of z. The predecessor function can be implemented in $\mathbb{VPC}$. The details can be found in Section 6. The behavior of $R(y)$ is captured by the following action sequence:

$$
R(\underline{n}) \xrightarrow{\tau} \top \xrightarrow{\overline{r}(\underline{n})} \top \overset{\tau}{\Longrightarrow}_{\underline{0} < \underline{n}} \xrightarrow{\overline{r}(\underline{n-1})} \top \overset{\tau}{\Longrightarrow}_{\underline{0} < \underline{n-1}} \cdots \overset{\tau}{\Longrightarrow}_{\underline{0} < \underline{1}} \xrightarrow{\overline{r}(\underline{1})} \top \cdots .
$$

It is obvious that $R(\underline{n})$ and $R(\underline{n'})$ are inequivalent whenever $n \neq n'$. The processes $S(y)$ and $T(y)$ are bisimilar since the action $S(y) \xrightarrow{\tau}_{\underline{0} \leq y} R(y)$ can be

$$
\begin{aligned}
U(y) &\stackrel{\text{def}}{=} \tau.W(y) + \mathit{if}\ \underline{0} \le y \le \underline{m}\ \mathit{then}\ \tau.R(y),\\
V(y) &\stackrel{\text{def}}{=} \tau.W(y) + \mathit{if}\ y = \underline{0}\ \mathit{then}\ \tau.R(y) + \ldots + \mathit{if}\ y = \underline{m}\ \mathit{then}\ \tau.R(y),\\
W(y) &\stackrel{\text{def}}{=} \mathit{if}\ \underline{0} = y\ \mathit{then}\ \tau.(\mathit{if}\ y = \underline{0}\ \mathit{then}\ \tau.R(y)\ \mathit{else}\ \overline{e}(\underline{0}))\\
&\quad + \mathit{if}\ \underline{0} \le y \le \underline{1}\ \mathit{then}\ \tau.(\mathit{if}\ y = \underline{1}\ \mathit{then}\ \tau.R(y)\ \mathit{else}\ \overline{e}(\underline{0}))\\
&\quad + \ldots\\
&\quad + \mathit{if}\ \underline{0} \le y \le \underline{m}\ \mathit{then}\ \tau.(\mathit{if}\ y = \underline{m}\ \mathit{then}\ \tau.R(y)\ \mathit{else}\ \overline{e}(\underline{0})).
\end{aligned}
$$

Fig. 5. Non-transitivity of $\simeq^?$

simulated by $T(y)$ as long as y is instantiated by a numeral. On the other hand it is easy to see that $S(y) \not\simeq^? T(y)$. The only way to simulate the action is by the collection $\{T(y) \stackrel{\tau}{\Longrightarrow}_{\underline{0}\le y\le \underline{n}}\ \mathit{if}\ y = \underline{n}\ \mathit{then}\ \tau.R(y)\ \mathit{else}\ \overline{e}(\underline{0})\}_{n\in\mathbf{N}}$. But notice that $\mathit{if}\ \underline{0} \le y \le \underline{n}\ \mathit{then}\ R(y)$ is *not* bisimilar to $\mathit{if}\ \underline{0} \le y \le \underline{n}\ \mathit{then}\ (\mathit{if}\ y = \underline{n}\ \mathit{then}\ \tau.R(y)\ \mathit{else}\ \overline{e}(\underline{0}))$. The relation $\simeq^?$ has to be abandoned because it is not transitive! Fig. 5 offers a counter example. One has $U(y) \simeq^? V(y)$ and $V(y) \simeq^? W(y)$ but not $U(y) \simeq^? W(y)$. It follows that $U(y) \not\simeq^? W(y)$.

The symbolic bisimulations need be defined in a more subtle way. The key idea of Hennessy and his collaborators is that the partition of the condition under which the simulated action is fired should not depend on the conditions under which the simulations are done. In their definition a partition of ψ is a collection $\{\psi_i\}_{i\in I}$ such that $\psi \Leftrightarrow \bigvee_{i\in I}\psi_i$, where the indexing set I could be as large as the size of the model. The reason that they may resort to the all-powerful operator $\bigvee$ is that they are using a meta logic about the oracle model. We will use a more restricted approach. We insist that the symbolic semantics of a value-passing calculus should only make use of the first order logic by which the logical theory of the calculus is defined.

Definition 9. *Suppose* Th *is a theory over* Σ, V *is a finite set of variables, and* $fv(\varphi) \subseteq V$. *A* boolean Th-partition *of* φ *on* V *is a finite set of boolean* Σ*-expressions* $\{\varphi_i\}_{i\in I}$ *such that* $fv(\bigvee_{i\in I}\varphi_i) \subseteq V$, $\mathsf{Th} \vdash \varphi_i \wedge \varphi_j \Rightarrow \bot$ *if* $i \neq j$, *and* $\mathsf{Th} \vdash \varphi \Leftrightarrow \bigvee_{i\in I}\varphi_i$.

Let $K(y) = (a)(\overline{a}(\underline{0}) \mid \mu X.a(x).(X \mid \mathit{if}\ x \le y\ \mathit{then}\ \overline{a}(\mathsf{s}(x))))$. It admits the following infinite sequence $K(y) \stackrel{\tau}{\longrightarrow}_{\top} \stackrel{\tau}{\longrightarrow}_{\underline{0}\le y} \stackrel{\tau}{\longrightarrow}_{\underline{1}\le y} \ldots \stackrel{\tau}{\longrightarrow}_{\underline{n}\le y} \ldots$. Although every finite subset of $\{\top, \underline{0} \le y, \underline{1} \le y, \ldots, \underline{n} \le y, \ldots\}$ is consistent, the infinite sequence is a fake. For every numeral $\underline{n}$ the process $K(\underline{n})$ terminates. From the point of view of the model $\mathbf{N}$, no assignment can satisfy all the Σ_{PA}-expressions in $\{\top, \underline{0} \le y, \underline{1} \le y, \ldots, \underline{n} \le y, \ldots\}$. From the point of view of the first order logic, no satisfiable Σ_{PA}-expression implies all the Σ_{PA}-expressions in $\{\top, \underline{0} \le y, \underline{1} \le y, \ldots, \underline{n} \le y, \ldots\}$. A faithful symbolic interpretation of divergence would make use of the infinite conjunction, which as we have argued, is not in line with the philosophy of the symbolic approach. To get a glimpse of the complication of codivergence, take a look at $H(y) = (a)(\overline{a}(\underline{0}) \mid \mu X.a(x).(X \mid \mathit{if}\ y \le x\ \mathit{then}\ \overline{a}(\mathsf{s}(x))))$, which is a slight modification of the previous $\mathbb{VPC}$-term. This term also admit an infinite sequence

of τ-actions $H(y) \xrightarrow{\tau} \top \xrightarrow{\tau} y \leq \underline{0} \xrightarrow{\tau} y \leq \underline{1} \ldots \xrightarrow{\tau} y \leq \underline{n} \ldots$. The infinite sequence is not real for $H(\underline{1}), H(\underline{2}), H(\underline{3}), \ldots$. But it is real for $H(\underline{0})$. These examples suggest that it is difficult to give a nice symbolic description of the codivergence property at a general level. It should be remarked though that divergence is intrinsically an undecidable property. If our interest in symbolic semantics is confined to automatic verification, the codivergence condition has to be dropped.

Definition 10. *A symmetric relation $\mathcal{R}$ on $\mathcal{T}_{\mathbb{VPC}_{\mathsf{Th}}}$ is a* symbolic bisimulation *if the following hold whenever $S\mathcal{R}T$ and $V = fv(S \,|\, T)$:*

1. *If $S \xrightarrow{\tau}_{\varphi} S'$ and $\mathsf{Th} \cup \{\varphi\}$ is consistent, then there are a boolean Th-partition $\{\varphi_i\}_{i \in I}$ of φ on V and a collection $\{T \Longrightarrow_{\psi_i} T_i\}_{i \in I}$ such that, for each $i \in I$, $\mathsf{Th} \vdash \varphi_i \Rightarrow \psi_i$ and $\varphi_i S \mathcal{R} \varphi_i T_i$, and one of the following properties holds:*
   (a) $\varphi_i S' \mathcal{R} \varphi_i T_i$;
   (b) $T_i \xrightarrow{\tau}_{\psi_i'} T_i'$ for some ψ_i', T_i' such that $\mathsf{Th} \vdash \varphi_i \Rightarrow \psi_i'$, and $\varphi_i S' \mathcal{R} \varphi_i T_i'$.
2. *If $S \xrightarrow{\overline{a}(t)}_{\varphi} S'$ and $\mathsf{Th} \cup \{\varphi\}$ is consistent, then there are a boolean Th-partition $\{\varphi_i\}_{i \in I}$ of φ on V and a collection $\{T \Longrightarrow_{\psi_i} T_i \xrightarrow{\overline{a}(t_i)}_{\psi_i'} T_i'\}_{i \in I}$ such that $\mathsf{Th} \vdash (\varphi_i \Rightarrow \psi_i \psi_i') \wedge (\varphi_i \Rightarrow t = t_i)$, and moreover $\varphi_i S \mathcal{R} \varphi_i T_i$ and $\varphi_i S' \mathcal{R} \varphi_i T_i'$ for every $i \in I$.*
3. *If $S \xrightarrow{a(x)}_{\varphi} S'$ and $\mathsf{Th} \cup \{\varphi\}$ is consistent, then there are a boolean Th-partition $\{\varphi_i\}_{i \in I}$ of φ on $V \cup \{x\}$ and a collection $\left\{T \Longrightarrow_{\psi_i} T_i \xrightarrow{a(x)}_{\psi_i'} T_i'\right\}_{i \in I}$ such that $\mathsf{Th} \vdash \varphi_i \Rightarrow \psi_i \psi_i'$, $\varphi_i S \mathcal{R} \varphi_i T_i$ and $\varphi_i S' \mathcal{R} \varphi_i T_i'$ for all $i \in I$.*

The symbolic bisimilarity $\simeq^s_{\mathsf{Th}}$ *is the largest symbolic bisimulation.*

In the above definition the requirement that $\mathsf{Th} \cup \{\varphi\}$ being consistent is important. Without the condition the $\mathbb{VPC}$-term *if* $x \neq x$ *then* $\overline{a}(\underline{0})$ would be wrongfully distinguished from $\mathbf{0}$.

The main algebraic properties of $\simeq^s_{\mathsf{Th}}$ are summarized in the next proposition.

Proposition 3. *The following statements about $\simeq^s_{\mathsf{Th}}$ are valid.*

1. *The relation $\simeq^s_{\mathsf{Th}}$ is an equivalence.*
2. *The relation $\simeq^s_{\mathsf{Th}}$ is a congruence.*
3. *If $S \simeq^s_{\mathsf{Th}} T$ then $S\sigma \simeq^s_{\mathsf{Th}} T\sigma$ for every substitution σ.*

Proof. (1) The proof of transitivity is routine and tedious. (2) The proof of the congruence property is standard after demonstrating that, for every boolean expression φ, $S \simeq^s_{\mathsf{Th}} T$ implies $\varphi S \simeq^s_{\mathsf{Th}} \varphi T$. (3) Using Lemma 3 it is easy to show that $\{(S\sigma, T\sigma) \mid S \simeq^s_{\mathsf{Th}} T\}$ is a symbolic bisimulation. We need to make use of a meta theoretical result asserting that $\mathsf{Th} \vdash \varphi\sigma$ for every substitution σ whenever $\mathsf{Th} \vdash \varphi$, which is an easy consequence of FO1 and FO2. □

The next technical lemma, whose simple proof is omitted, helps understand the above definition. It is the symbolic version of the Stuttering Lemma of van Glabbeek and Weijland [vGW89].

Lemma 6. *Suppose* $T \xrightarrow{\tau}_{\varphi_1} T_1 \xrightarrow{\tau}_{\varphi_2} T_2 \ldots \xrightarrow{\tau}_{\varphi_n} T_n$ *and* $\varphi \Rightarrow \varphi_1 \ldots \varphi_n$. *Then* $\varphi T \simeq^s_{\mathsf{Th}} \varphi T_n$ *implies* $\varphi T_1 \simeq^s_{\mathsf{Th}} \varphi T_2 \simeq^s_{\mathsf{Th}} \ldots \simeq^s_{\mathsf{Th}} \varphi T_n$.

Now let's take a look at some examples. Consider the following $\mathbb{VPC}$-terms:

$$\begin{aligned} E(y) &\stackrel{\text{def}}{=} \mu X.\mathit{if}\ y = \underline{0} \vee y = \underline{1}\ \mathit{then}\ \overline{b}(y).X, \\ A(y) &\stackrel{\text{def}}{=} E(y) \,|\, \mathit{if}\ y = \underline{0}\ \mathit{then}\ \overline{b}(y), \\ B(y) &\stackrel{\text{def}}{=} E(y) \,|\, \mathit{if}\ y = \underline{1}\ \mathit{then}\ \overline{b}(y). \end{aligned}$$

It is clear that $A(y) \simeq^s_{\mathsf{Th}} B(y)$. The action $A(y) \xrightarrow{\overline{b}(y)}_{y=\underline{0}} E(y)$ is simulated by the single action $B(y) \xrightarrow{\overline{b}(y)}_{y=\underline{0}\vee y=\underline{1}} E(y) \,|\, \mathit{if}\ y = \underline{1}\ \mathit{then}\ \overline{b}(y)$. This example shows that the condition $\mathsf{Th} \vdash \varphi_i \Rightarrow \psi_i$ in say (1) of Definition 10 should not be strengthened to $\mathsf{Th} \vdash \varphi_i \Leftrightarrow \psi_i$.

The finiteness of partition is a genuine restriction. Let's see a counter example. Consider the following $\mathbb{VPC}$-terms:

$$\begin{aligned} F(y) &\stackrel{\text{def}}{=} \mu X.a(x).(\overline{a}(\mathsf{s}(x)) \,|\, \mathit{if}\ y{=}x\ \mathit{then}\ \overline{b}(x)\ \mathit{else}\ X), \\ C(y) &\stackrel{\text{def}}{=} \mathit{if}\ \underline{0} \leq y\ \mathit{then}\ \overline{b}(y), \\ D(y) &\stackrel{\text{def}}{=} (a)(\overline{a}(\underline{0}) \,|\, F(y)). \end{aligned}$$

Typically D has the action sequence $D(y) \stackrel{\tau}{\Longrightarrow}_{y\neq\underline{0}\wedge\ldots\wedge y\neq\underline{n-1}} \xrightarrow{\overline{b}(\underline{n})}_{y=\underline{n}}$. It is not difficult to see that $C(y) =_{\mathsf{Th}} D(y)$. The action $C(y) \xrightarrow{\overline{b}(y)}_{\underline{0}\leq y} \mathbf{0}$ is simulated by the *infinite* collection $\{D(y) \stackrel{\tau}{\Longrightarrow}_{y\neq\underline{0}\wedge\ldots\wedge y\neq\underline{n-1}} \xrightarrow{\overline{b}(\underline{n})}_{y=\underline{n}}\}_{n\in\mathbf{N}}$. There is no finite collection that can simulate $C(y) \xrightarrow{\overline{b}(y)}_{\underline{0}\leq y} \mathbf{0}$. So the symbolic bisimilarity $\simeq^s_{\mathsf{Th}}$ cannot coincide with the absolute equality $=_{\mathsf{Th}}$ even if divergence is ignored. It is however a correct approximation of the absolute equality.

Theorem 2. *Let* S, T *be finite* $\mathbb{VPC}_{\mathsf{Th}}$*-terms. Then* $S =_{\mathsf{Th}} T$ *if* $S \simeq^s_{\mathsf{Th}} T$.

Proof. Let $\mathcal{R}$ be the following binary relation

$$\left\{ (S\rho, T\rho) \,\middle|\, \begin{array}{l} S, T \text{ are finite, } S \simeq^s_{\mathsf{Th}} T \text{ and} \\ \rho \text{ is an assignment such that } bv(S \,|\, T) \cap dom(\rho) = \emptyset \end{array} \right\}.$$

Suppose $S \simeq^s_{\mathsf{Th}} T$ and $S\rho \xrightarrow{a(t)} P$ for some $t \in \mathsf{T}^0_\Sigma$ and some assignment ρ such that $bv(S \,|\, T) \cap dom(\rho) = \emptyset$. Let V be $fv(S \,|\, T)$. By Lemma 5, $S\rho \xrightarrow{a(x)}_{\varphi'} S'$ for some Th-theorem φ' and some x, S' such that $P \equiv S'\{t/x\}$. By Lemma 3, there exist some φ, S_1 such that $\varphi' \equiv \varphi\rho$, $S' \equiv S_1\rho$ and $S \xrightarrow{a(x)}_{\varphi} S_1$. By the definition of symbolic bisimulation there are a boolean Th-partition $\{\varphi_i\}_{i\in I}$ of φ on $V \cup \{x\}$ and a collection $\left\{ T \Longrightarrow_{\psi_i} T_i \xrightarrow{a(x)}_{\psi'_i} T'_i \right\}_{i\in I}$ such that $\mathsf{Th} \vdash \varphi_i \Rightarrow \psi_i\psi'_i$,

$\varphi_i S \mathcal{R} \varphi_i T_i$ and $\varphi_i S' \mathcal{R} \varphi_i T_i'$ for all $i \in I$. Now $\mathsf{Th} \vdash \varphi_i \rho[x \leftarrow t]$ for some $i \in I$, which in turn implies $\mathsf{Th} \vdash \psi_i \psi_i' \rho[x \leftarrow t]$, $S\rho \mathcal{R} T_i \rho$ and $S'\rho[x \leftarrow t] \mathcal{R} T_i'\rho[x \leftarrow t]$. At the meantime notice that $T\rho \Longrightarrow T_i\rho \xrightarrow{a(t)} T_i'\rho[x \leftarrow t]$ by Lemma 3 and Lemma 4. Conclude that $\mathcal{R}$ is a bisimulation. The relation is also extensional by Proposition 3. It is equipollent since the external actions are bisimulated. □

5 Proof System

When confined to the finite $\mathbb{VPC}_{\mathsf{Th}}$-processes, one expects that the equality can be mechanically checked. Often such an algorithm is based on a complete equational system. For most value-passing calculi defined in literature a self-contained decidable procedure for equivalence checking is out of the question since the logics/models are undecidable. For example, to check if $\varphi\overline{a}(t).S = \varphi\overline{a}(t').S$ holds, one has to check if $\varphi \Rightarrow t = t'$ is valid. An algorithm for checking the equivalence of finite processes has to make use of an oracle that answers every question on the validity of a logical formula. For a value-passing calculus $\mathbb{VPC}_{\mathsf{Th}}$ studied in this paper, equivalence checking algorithms do exist.

In studying proof systems, a standard treatment is to remove the composition operator using *expansion law* [HM85, Mil89a]. This is done at the expense of introducing the unguarded choice operator '+' whose semantics is defined by

$$\frac{S \xrightarrow{\lambda}_{\varphi} S'}{S + T \xrightarrow{\lambda}_{\varphi} S'} \qquad \frac{T \xrightarrow{\lambda}_{\varphi} T'}{S + T \xrightarrow{\lambda}_{\varphi} T'}$$

The operator destroys the congruence property of process equalities. As a consequence additional complication is introduced to produce a congruence. We avoid the complication by not using the congruence relation. In this section the notation $\sum_{i \in I} \varphi_i \lambda_i.S_i$ stands for a mixed choice [Pal03, FL10].

Suppose $S \equiv \sum_{i \in I} \varphi_i \lambda_i.S_i$ and $T \equiv \sum_{j \in J} \psi_j \lambda_j.T_j$. The expansion law is formulated by the following equality:

$$\begin{aligned} S \,|\, T = &\sum_{i \in I} \varphi_i \lambda_i.(S_i \,|\, T) + \sum_{\lambda_j = \overline{a}(t_j)}^{\lambda_i = a(x)} \varphi_i \psi_j \tau.(S_i\{t_j/x\} \,|\, T_j) \\ &+ \sum_{j \in J} \psi_j \lambda_j.(S \,|\, T_j) + \sum_{\lambda_i = \overline{b}(t_i)}^{\lambda_j = b(y)} \varphi_i \psi_j \tau.(S_i \,|\, T_j\{t_i/y\}). \end{aligned}$$

Our equational system AS_{Th} consists of the first order logic axioms defined in Fig. 1, the nonlogical axioms of Th, the equational axioms defined in Fig. 6 and the expansion law. We write $AS_{\mathsf{Th}} \vdash S = T$ if the equality $S = T$ can be derived within AS_{Th}, and $S \overset{R}{=} T$ if R is the major law used to derive the equality $S = T$.

Most of the laws are variants of the axioms proposed in previous studies on the value-passing calculi [IL01] and the name-passing calculi [PS95]. The law $S6$ is a generalization of the following law proposed by Parrow and Sangiorgi [PS95]:

$$a(x).S + a(x).T = a(x).S + a(x).T + a(x).(\varphi S + \neg\varphi T). \tag{2}$$

$S1$	$T + \mathbf{0} = T$	
$S2$	$S + T = T + S$	
$S3$	$R + (S + T) = (R + S) + T$	
$S4$	$T + T = T$	
$S5$	$\varphi T + \varphi' T = (\varphi \vee \varphi')T$	
$S6$	$\sum_{i\in I} \phi_i a(x).(\neg\varphi_i T_i + \varphi_i \tau.T) = \varphi a(x).T + \sum_{i\in I} \phi_i a(x).(\neg\varphi_i T_i + \varphi_i \tau.T)$	if $\mathsf{Th} \vdash \varphi_i \Rightarrow \phi_i$ for all $i \in I$ and $\mathsf{Th} \vdash \varphi \Leftrightarrow \bigvee_{i\in I} \varphi_i$.
$C1$	$[\bot]T = \mathbf{0}$	
$C2$	$[\top]T = T$	
$C3$	$\varphi\overline{a}(t).T = \varphi\overline{a}(t').T$,	if $\mathsf{Th} \vdash \varphi \Rightarrow t = t'$
$C4$	$\varphi\lambda.T = \varphi\lambda.\varphi T$	
$C5$	$\phi(T + T') = \phi T + \phi T'$,	
$C6$	$\varphi T = \psi T$,	if $\mathsf{Th} \vdash \varphi \Leftrightarrow \psi$
$C7$	$\varphi(\psi T) = (\varphi\psi)T$	
$L1$	$(c)\mathbf{0} = \mathbf{0}$	
$L2$	$(c)\lambda.T = \mathbf{0}$,	if c is in λ
$L3$	$(c)\lambda.T = \lambda.(c)T$,	if c is not in λ
$L4$	$(c)(T + T') = (c)T + (c)T'$	
$L5$	$(c)\varphi T = \varphi(c)T$	
B	$\lambda.(S + T) = \lambda.(\tau.(S + T) + T)$	

Fig. 6. Axiom for Finite $\mathbb{VPC}_{\mathsf{Th}}$-Term

As pointed out in [PS95], this is the law that tells apart the early equivalence and the late equivalence [MPW92]. The combination of the S-laws has powerful consequences, two of which are given by the following lemmas.

Lemma 7. *$AS_{Th} \vdash \psi\lambda.T = \varphi\lambda.T + \psi\lambda.T$ if $\mathsf{Th} \vdash \varphi \Rightarrow \psi$.*

Lemma 8. *$AS_{Th} \vdash \varphi(\lambda.T)\{t/x\} = \varphi(\lambda.T)\{t'/x\}$ if $\mathsf{Th} \vdash \varphi \Rightarrow t = t'$.*

The B-law is due to van Glabbeek and Weijland [vGW89]. It implies Milner's first tau law $\lambda.\tau.T = \lambda.T$ and a weaker form $\tau.(T + \varphi\tau.T) = \tau.T$ of Milner's second tau law [Mil89a].

The verification of soundness property of AS_{Th} is routine.

Lemma 9. *If $AS_{Th} \vdash S = T$ then $S \simeq^s_{Th} T$.*

A nice property of a proof system is that it allows one to focus on terms in some special form.

Definition 11. *A finite $\mathbb{VPC}_{Th}$-term is a* normal form *if it is either $\mathbf{0}$ or of the form $\sum_{i\in I} \varphi_i\lambda_i.T_i$ such that T_i is a normal form for every $i \in I$.*

Using $L4$ and $L5$ one can pull all the localization operations in a term to the very front. Then one may remove all the composition operations by applying the expansion law. And finally one removes the localization operations using the L-axioms. What one gets is a normal form term. Hence the following lemma.

Lemma 10. *For each finite $\mathbb{VPC}_{\mathsf{Th}}$-term T there is some normal form T' such that $AS_{\mathsf{Th}} \vdash T = T'$.*

In the rest of the section the following Completeness Theorem is proved.

Theorem 3. *$S \simeq^s_{\mathsf{Th}} T$ if and only if $AS_{\mathsf{Th}} \vdash \tau.S = \tau.T$.*

Proof. A sequence $T \stackrel{\tau}{\Longrightarrow}_{\varphi} T'$ is *maximal* with regards to ϕ if (i) $\mathsf{Th} \vdash \phi \Rightarrow \varphi$, (ii) $\phi T \simeq^s_{\mathsf{Th}} \phi T''$, and (iii) there does not exist any T'' such that $T' \stackrel{\tau}{\longrightarrow}_{\varphi'} T''$, $\mathsf{Th} \vdash \phi \Rightarrow \varphi'$ and $\phi T \simeq^s_{\mathsf{Th}} \phi T''$. Intuitively $T \stackrel{\tau}{\Longrightarrow}_{\varphi} T'$ is maximal if T' cannot evolve to another state that stays equal to T under the condition ϕ.

We confine our attention to the normal forms. Suppose $S \simeq^s_{\mathsf{Th}} T$ and S, T are the normal forms $\sum_{i\in I} \varphi_i\lambda_i.S_i$ and $\sum_{j\in J} \psi_j\lambda_j.T_j$ respectively. We shall prove that the following properties hold for the normal forms:

(S) If $\mathsf{Th} \vdash \phi \Rightarrow \varphi$, $\phi T \simeq^s_{\mathsf{Th}} \phi T'$ and $T \stackrel{\tau}{\Longrightarrow}_{\varphi} T'$ is maximal with regards to ϕ, then $AS_{\mathsf{Th}} \vdash \tau.T = \tau.(T + \phi T')$.
(P) If $\phi T \simeq^s_{\mathsf{Th}} \phi S$, then $AS_{\mathsf{Th}} \vdash \phi\tau.T = \phi\tau.(T + S) = \phi\tau.S$.

Let's write $dep(T)$ for the maximal nested depth of prefixing operation of T. Our proof strategy will be as follows:

1. Prove (S) assuming that (S) and (P) hold for terms with depths less than i.
2. Prove (P) assuming that (P) holds for terms whose depths are less than i and that (S) holds for terms whose depths are less than or equal to i.

The inductive proof of (S) is given as follows: Suppose T is a normal form of depth i and $T \stackrel{\tau}{\longrightarrow}_{\varphi_1} T_1 \stackrel{\tau}{\longrightarrow}_{\varphi_2} \ldots T_{n-1} \stackrel{\tau}{\longrightarrow}_{\varphi_n} T_n$ is maximal with regards to φ with $\varphi T_1 \simeq^s_{\mathsf{Th}} \varphi T_n$ and $\varphi = \varphi_1 \ldots \varphi_n$. Then $\varphi T_1 \simeq^s_{\mathsf{Th}} \varphi T_2 \simeq^s_{\mathsf{Th}} \ldots \simeq^s_{\mathsf{Th}} \varphi T_n$ by Lemma 6. By the induction hypothesis on (P), $AS_{\mathsf{Th}} \vdash \varphi\tau.T_1 = \varphi\tau.T_n$. By Lemma 7, $AS_{\mathsf{Th}} \vdash \tau.T = \tau.(T + \varphi_1\tau.T_1) = \tau.(T + \varphi\tau.T_1) = \tau.(T + \varphi\tau.T_n)$. Since $T \stackrel{\tau}{\longrightarrow}_{\varphi_1} T_1 \stackrel{\tau}{\longrightarrow}_{\varphi_2} \ldots T_{n-1} \stackrel{\tau}{\longrightarrow}_{\varphi_n} T_n$ is maximal, the action of each summand $\psi_j\lambda_j.T_j$ of T can be simulated by a set of summands of T_n using induction hypothesis, and consequently $\psi_j\lambda_j.T_j$ can be assimilated by T_n. We conclude that $AS_{\mathsf{Th}} \vdash \varphi\tau.T_n = \varphi\tau.(T_n + T)$. Consequently

$$\begin{aligned}
AS_{\mathsf{Th}} \vdash \tau.T &= \tau.(T + \varphi\tau.(T_n + T)) \\
&\stackrel{S5}{=} \varphi\tau.(T + \varphi\tau.(T_n + T)) + \neg\varphi\tau.(T + \varphi\tau.(T_n + T)) \\
&\stackrel{C \text{ laws}}{=} \varphi\tau.(T + \tau.(T_n + T)) + \neg\varphi\tau.T \\
&\stackrel{B}{=} \varphi\tau.(T + T_n) + \neg\varphi\tau.(T + \varphi T_n) \\
&\stackrel{S5}{=} \varphi\tau.(T + \varphi T_n) + \neg\varphi\tau.(T + \varphi T_n) \\
&\stackrel{S5}{=} \tau.(T + \varphi T_n).
\end{aligned}$$

We turn to the inductive proof of (P). By the C-laws we only have to prove the special case when $\phi = \top$. Consider a summand $\varphi_i\lambda_i.S_i$ of S. The action

$\sum_{i\in I}\varphi_i\lambda_i.S_i \xrightarrow{\lambda_i}_{\varphi_i} S_i$ must be simulated by the term $\sum_{j\in J}\psi_j\lambda_j.T_j$. There are three cases according to the shape of λ_i. We only consider the case $\lambda_i = a(x)$. By definition there are a boolean Th-partition $\{\varphi_i^k\}_{k\in K}$ of φ_i on $fv(S\,|\,T)\cup\{x\}$ and a collection $\{T \Longrightarrow_{\psi_k} T_k \xrightarrow{a(x)}_{\psi'_k} T'_k\}_{k\in K}$ such that $\mathsf{Th}\vdash \varphi_i^k \Rightarrow \psi_k\psi'_k$ for all $k\in K$, and $\varphi_i^k S \simeq^s_{\mathsf{Th}} \varphi_i^k T_k$, $\varphi_i^k S_i \simeq^s_{\mathsf{Th}} \varphi_i^k T'_k$ for every $k\in K$. We assume that for each $k\in K$, $T\Longrightarrow_{\psi_k} T_k$ is maximal with regards to φ_i^k. By induction on (P),

$$AS_{\mathsf{Th}} \vdash \varphi_i^k\tau.S = \varphi_i^k\tau.T_k, \tag{3}$$

$$AS_{\mathsf{Th}} \vdash \varphi_i^k\tau.S_i = \varphi_i^k\tau.T'_k \tag{4}$$

for every $k\in K$. Since $\{T+S \Longrightarrow_{\psi_k} T_k \xrightarrow{a(x)}_{\psi'_k} T'_k\}_{k\in K}$, we could have the following rewriting that makes use of the inductive hypothesis on (S).

$$\begin{aligned}
\tau.(T+S) &\overset{I,H.}{=} \tau.\left(T+S+\sum_{k\in K}\psi_k T_k\right)\\
&= \tau.\left(T+S+\sum_{k\in K}\psi_k(T_k+\psi'_k a(x).T'_k)\right)\\
&\overset{C5}{=} \tau.\left(T+S+\sum_{k\in K}\psi_k T_k+\sum_{k\in K}\psi_k\psi'_k a(x).T'_k\right)\\
&\overset{B}{=} \tau.\left(T+S+\sum_{k\in K}\psi_k T_k+\sum_{k\in K}\psi_k\psi'_k a(x).\tau.T'_k\right)\\
&\overset{S5}{=} \tau.\left(T+S+\sum_{k\in K}\psi_k T_k+\sum_{k\in K}\psi_k\psi'_k a(x).(\neg\varphi_i^k\tau.T'_k+\varphi_i^k\tau.T'_k)\right)\\
&\overset{(4)}{=} \tau.\left(T+S+\sum_{k\in K}\psi_k T_k+\sum_{k\in K}\psi_k\psi'_k a(x).(\neg\varphi_i^k\tau.T'_k+\varphi_i^k\tau.S_i)\right)\\
&\overset{S6}{=} \tau.\left(T+\sum_{i'\in I\setminus\{i\}}\varphi_{i'}\lambda_{i'}.S_{i'}+\sum_{k\in K}\psi_k T_k+\sum_{k\in K}\psi_k\psi'_k a(x).(\neg\varphi_i^k\tau.T'_k+\varphi_i^k\tau.S_i)\right)\\
&= \tau.\left(T+\sum_{i'\in I\setminus\{i\}}\varphi_{i'}\lambda_{i'}.S_{i'}+\sum_{k\in K}\psi_k T_k\right)\\
&\overset{I,H.}{=} \tau.\left(T+\sum_{i'\in I\setminus\{i\}}\varphi_{i'}\lambda_{i'}.S_{i'}\right).
\end{aligned}$$

It follows from induction that $\tau.(T+S) = \tau.\left(T+\sum_{i\in I'}\phi_i\tau.T\right)$ for some $I'\subseteq I$ and soem $\{\phi_i\}_{i\in I'}$. Let $\phi = \bigvee_{i\in I'}\phi_i$. Then

$$\begin{aligned}
\tau.(T+S) &\overset{S5}{=} \tau.(T+\phi\tau.T)\\
&\overset{S5,C2}{=} \phi\tau.(T+\phi\tau.T)+\neg\phi\tau.(T+\phi\tau.T)\\
&\overset{C\ \text{laws},B}{=} \phi\tau.T+\neg\phi\tau.T\\
&\overset{S5,C2}{=} \tau.T.
\end{aligned}$$

Symmetrically $AS_{\mathsf{Th}}\vdash\tau.(S+T)=\tau.S$. We are done. □

6 Turing Completeness

The value-passing calculi are rudimentary process models. The question concerning their expressiveness has to be settled. Which value-passing calculi are for instance complete in the sense that all recursive functions [Rog87] are definable? To answer the question we need to make it clear how the natural numbers are defined in a value-passing calculus. From the operational point of view, a natural number system is not just an infinite set of pairwise distinct closed Σ-terms, but also an effective way of generating these Σ-terms.

Definition 12. *A* numeric system *for* $\mathbb{VPC}_{\mathsf{Th}}$ *consists of a countable subset* $\{\widehat{0}, \widehat{1}, \widehat{2}, \ldots, \widehat{n}, \ldots\}$ *of* $\mathcal{T}^0_\Sigma$ *and a* $\mathbb{VPC}_{\mathsf{Th}}$*-term* $S_d(x)$ *that satisfy the followings:*

1. *The variable* x *is the only free variable appearing in* $S_d(x)$.
2. $\mathsf{Th} \vdash \widehat{p} \neq \widehat{q}$ *for all* $p, q \in \mathbf{N}$ *such that* $p \neq q$.
3. *Every action sequence of* $S_d(\widehat{n})$ *is of the form* $S_d(\widehat{n}) \Longrightarrow \stackrel{\overline{d}(\widehat{n+1})}{\longrightarrow} =_{\mathsf{Th}} \mathbf{0}$.

It is clear from (3) of Definition 12 that every action sequence of $S_d(\widehat{n})$ is actually of the form $S_d(\widehat{n}) \rightarrow^* \stackrel{\overline{d}(\widehat{n+1})}{\longrightarrow} =_{\mathsf{Th}} \mathbf{0}$ and $S_d(\widehat{n}) =_{\mathsf{Th}} \overline{d}(\widehat{n+1})$.

Using a numeric system, we may talk about functions in a value-passing calculus. The *predecessor function* p for instance can be defined in such a calculus. Suppose we would like to have the $\mathbb{VPC}$-process $a(x).\overline{b}(\mathsf{p}(x))$. It can be defined by $a(x).(c)(\overline{c}(\underline{0}) \mid !c(y).\mathit{if}\ x = \mathsf{s}(y)\ \mathit{then}\ \overline{b}(y)\ \mathit{else}\ \overline{c}(\mathsf{s}(y)))$. The process diverges when given the input $\underline{0}$. As is demonstrated by this example, each application of the predecessor function is implemented by an additional $\mathbb{VPC}$-term. In sequel we shall make use of the predecessor function without worrying about how a particular occurrence of the function is implemented.

An n-ary partial function $f(x_1, \ldots, x_n)$ is *definable* in $\mathbb{VPC}_{\mathsf{Th}}$ with respect to the numeric system $\langle\{\widehat{0}, \widehat{1}, \widehat{2}, \ldots, \widehat{n}, \ldots\}, S_d(x)\rangle$ if, for all names $a_1, \ldots, a_n, b$, there is a process $I(a_1, \ldots, a_n, b)$ of the form $a_1(x_1)....a_n(x_n).T$ such that (i) if $f(p_1, \ldots, p_n)$ is defined, then all the action sequences of $T\{\widehat{p_1}/x_1, \ldots, \widehat{p_n}/x_n\}$ are finite and are of the following form $T\{\widehat{p_1}/x_1, \ldots, \widehat{p_n}/x_n\} \Longrightarrow \stackrel{\overline{b}(\widehat{p})}{\longrightarrow} T' =_{\mathsf{Th}} \mathbf{0}$, where $p = f(p_1, \ldots, p_n)$; and (ii) if $f(p_1, \ldots, p_n)$ is undefined, then $T\{\widehat{p_1}/x_1, \ldots, \widehat{p_n}/x_n\}$ can only perform τ-action sequences and all its τ-action sequences are divergent. We say that $f(x_1, \ldots, x_n)$ is defined by $I(a_1, \ldots, a_n, b)$ at $a_1, \ldots, a_n, b$. A set of partial functions is definable with respect to a numeric system if every member of the set is definable with respect to the numeric system. A set of partial functions is definable in $\mathbb{VPC}_{\mathsf{Th}}$ if it is definable with respect to some numeric system of $\mathbb{VPC}_{\mathsf{Th}}$.

If a function is definable in $\mathbb{VPC}_{\mathsf{Th}}$ with respect to $\langle\{\widehat{0}, \widehat{1}, \widehat{2}, \ldots, \widehat{n}, \ldots\}, S_d(x)\rangle$, we can design a procedure that, upon receiving the natural numbers $p_1, \ldots, p_n$, traverses the derivation tree of $T\{\widehat{p_1}/x_1, \ldots, \widehat{p_n}/x_n\}$. This is well defined since every $\mathbb{VPC}_{\mathsf{Th}}$-term is finite branching. The procedure terminates with the result p if $T\{\widehat{p_1}/x_1, \ldots, \widehat{p_n}/x_n\}$ export $\widehat{p}$ at some b. It diverges otherwise. According to the Church-Turing Thesis, this procedure defines a recursive function. We conclude that all functions definable in a value-passing calculus are recursive.

The opposite question asks if all the recursive functions can be defined in a value-passing calculus. This amounts to asking if the value-passing calculus is Turing complete.

Definition 13. *A value-passing calculus* $\mathbb{VPC}_{Th}$ *is* Turing complete *if all the recursive functions are definable in* $\mathbb{VPC}_{Th}$.

The next proposition provides a basic fact about the expressiveness of the value-passing calculi.

Proposition 4. *A value-passing calculus* $\mathbb{VPC}_{Th}$ *is Turing complete if and only if it has a numeric system.*

Proof. The implication in one direction is clear. Now suppose $\mathbb{VPC}_{\mathsf{Th}}$ has a numeric system $\langle\{\widehat{0},\widehat{1},\widehat{2},\ldots,\widehat{n},\ldots\}, S_d(x)\rangle$. We show that the recursive functions [Rog87] are definable in $\mathbb{VPC}_{\mathsf{Th}}$. We consider only two cases.

- Suppose $G(c_1,\ldots,c_n,d,e,b))$ and $H(c_1,\ldots,c_n,b)$ are the interpretations of two recursive functions. The interpretation $F(a_1,\ldots,a_{n+1},b)$ of the *recursion function* at $a_1,..,a_n,a_{n+1},b$ is

  $$a_1(x_1).\cdots.a_{n+1}(x_{n+1}).(f)(\overline{f}(\widehat{0},\widehat{1},x_{n+1},\widehat{0}) \mid (\widetilde{c})(!\overline{c_1}(x_1) \mid \cdots \mid !\overline{c_n}(x_n) \mid Rec)),$$

  where *Rec* stands for the following process

  $$\begin{array}{l} !f(y,y',z,v).\mathit{if}\ \widehat{0}=y=z\ \mathit{then}\ H(c_1,\ldots,c_n,b) \\ \quad \mathit{else\ if}\ \widehat{0}=y\wedge y'\leq z\ \mathit{then}\ (e)(H(c_1,\ldots,c_n,e)\mid e(v).\overline{f}(\widehat{0},\widehat{1},z,v)) \\ \quad\quad \mathit{else\ if}\ \widehat{0}<y\wedge y'=z\ \mathit{then}\ (d)(d')(G(c_1,\ldots,c_n,d,d',b)\mid \overline{d}(y)\mid \overline{d'}(v)) \\ \quad\quad\quad \mathit{else\ if}\ \widehat{0}\leq y\wedge y'<z\ \mathit{then}\ (d)(d')(e)(G(c_1,\ldots,c_n,d,d',e)\mid \overline{d}(y)\mid \overline{d'}(v) \\ \quad\quad\quad\quad \mid e(v).\overline{f}(S_d(y),S_d(y'),z,v). \end{array}$$

- Suppose $G(a_1,\cdots,a_{n+1},b)$ is the interpretation of a recursive function. The *minimalization function* is interpreted at $a_1,..,a_n,a_{n+1},b$ by the process

  $$a_1(x_1).\cdots.a_{n+1}(x_{n+1}).(a_1\ldots a_{n+1})(!\overline{a_1}(x_1) \mid \cdots \mid !\overline{a_n}(x_n) \mid \overline{f}(\widehat{0}) \mid !f(z).Mu),$$

  where Mu is the following process

  $$\overline{a_{n+1}}(z) \mid (c)(G(a_1,\cdots,a_{n+1},c) \mid c(v).\mathit{if}\ v=\widehat{0}\ \mathit{then}\ \overline{b}(z)\ \mathit{else}\ \overline{f}(S_d(z))).$$

We are done. □

Proposition 4 adds weight to Definition 12 since a numeric system is intuitively a minimal requirement for a value-passing calculus to simulate all the recursive functions. The $\mathbb{VPC}_{\mathsf{Th}}$-term $S_d(x)$ is nothing but an implementation of the successor function.

7 VPC and Recursion Theory

The previous three sections have developed a rigorous theory for the value-passing calculi. When applied to a particular decidable first order theory, this

general theory immediately generates an operational semantics and an observation theory for that calculus. We have studied only one decidable first order theory, the theory PA defined in Fig. 2. So in this section we take a closer look at the value-passing calculus defined on top of PA. We will write $=_{\mathbb{VPC}}$ for the absolute equality of $\mathbb{VPC}$ and $\simeq^{s}_{\mathbb{VPC}}$ for the symbolic bisimilarity of $\mathbb{VPC}$.

According to our general setting, $\mathbb{VPC}$ should have a number of virtues. Let's summarize its key features:

- $\mathbb{VPC}$ is Turing complete. So it is among the good value-passing calculi.
- The validity of the boolean propositions is known to be decidable. This is the reason for us to see $\mathbb{VPC}$ as a programming language. Our familiarity with the standard model $\mathbf{N}$ helps confidence in programming with $\mathbb{VPC}$.
- The simplicity of our Peano theory offers a nice algebraic theory of $\mathbb{VPC}$. Formal comparison of $\mathbb{VPC}$ against other well known process calculi is not only possible, but also instructive.

It is difficult to think of a value-passing calculus that is weaker than $\mathbb{VPC}$ but is still expressive enough. We now elaborate on this point.

To start with observe that the binary relation $<$ is not absolutely necessary for $\mathbb{VPC}$. The following proposition explains why.

Proposition 5. *For each $\mathbb{VPC}$-process P there is some $\mathbb{VPC}$-process P' such that $P =_{\mathbb{VPC}} P'$ and that P' contains no occurrence of the relation symbol $<$.*

Proof. The basic idea is that the boolean value of a closed atomic Σ_{PA}-expression $t < t'$ can be calculated within $\mathbb{VPC}$. Given a $\mathbb{VPC}$-process P, we can translate it into an equal $\mathbb{VPC}$-process P'. The encoding is structural on composition, localization and replication operators. The interpretation of the guarded choice and the conditional are similar. We take a look at how the latter is translated. Consider the $\mathbb{VPC}$-term $S \stackrel{\text{def}}{=} \mathit{if}\ \varphi\ \mathit{then}\ T$. The interpretation of S is given by the following term

$$(c)([\![\varphi]\!]_c \mid c(z).\mathit{if}\ z = \underline{1}\ \mathit{then}\ T')$$

where T' is the interpretation of T and $[\![\varphi]\!]_c$ is structurally defined as follows:

- If φ is $t = t'$, then $[\![\varphi]\!]_c$ is $\mathit{if}\ t = t'\ \mathit{then}\ \overline{c}(\underline{1})\ \mathit{else}\ \overline{c}(\underline{0})$.
- If φ is $t < t'$, then $[\![\varphi]\!]_c$ is the following term

$$\overline{d}(t).\overline{e}(t') \mid !d(x).e(y).\mathit{if}\ y = \underline{0}\ \mathit{then}\ \overline{c}(\underline{0})\ \mathit{else}\ \mathit{if}\ x = \underline{0}\ \mathit{then}\ \overline{c}(\underline{1})\ \mathit{else}\ \overline{d}(\mathsf{p}(x)).\overline{e}(\mathsf{p}(y)).$$

- If φ is $\varphi' \wedge \varphi''$, then $[\![\varphi]\!]_c$ is

$$(de)([\![\varphi']\!]_d \mid [\![\varphi'']\!]_e \mid d(y).e(z).\mathit{if}\ y = 1 \wedge z = 1\ \mathit{then}\ \overline{c}(\underline{1})\ \mathit{else}\ \overline{c}(\underline{0})).$$

- If φ is $\varphi' \vee \varphi''$, then $[\![\varphi]\!]_c$ is

$$(de)([\![\varphi']\!]_d \mid [\![\varphi'']\!]_e \mid d(y).e(z).\mathit{if}\ y = 1 \vee z = 1\ \mathit{then}\ \overline{c}(\underline{1})\ \mathit{else}\ \overline{c}(\underline{0})).$$

The equality $P =_{\mathbb{VPC}} P'$ holds in an obvious way. □

7.1 Minimality of VPC

In this section we prove that $\mathbb{VPC}$ plays an authentic role in all the Turing complete value-passing calculi. Suppose $\langle\{\widehat{0},\widehat{1},\widehat{2},\ldots,\widehat{n},\ldots\},S_d(x)\rangle$ is a numeric system of a Turing complete model $\mathbb{VPC}_{\mathsf{Th}}$. One could define a translation $[\![\_]\!]_{\mathsf{Th}}$ from $\mathbb{VPC}$ to $\mathbb{VPC}_{\mathsf{Th}}$ using the ideas described in Fig. 7. The translation of the action labels $[\![\lambda]\!]_{\mathsf{Th}}$ can be defined by

$$[\![\lambda]\!]_{\mathsf{Th}} \stackrel{\text{def}}{=} \begin{cases} \tau, & \text{if } \lambda=\tau,\\ a(\widehat{n}), & \text{if } \lambda=a(\underline{n}),\\ \overline{a}(\widehat{n}), & \text{if } \lambda=\overline{a}(\underline{n}).\end{cases}$$

Three aspects of the encoding call for explanation.

- One is that we have identified the set of the term variables of $\mathbb{VPC}_{\mathsf{PA}}$ with the term variables of $\mathbb{VPC}_{\mathsf{Th}}$.
- The second is that the operation $D_c(\_)$ is defined as follows: for a term $t\equiv \mathsf{s}^k(x)$ with $k>0$, $D_c(t)$ is the following term

  $$(c_0)(\overline{c_0}(x)\,|\,c_0(z_0).(c_1)(S_{c_1}(z_0)\,|\,c_1(z_1).(\ldots c_{k-1}(z_{k-1}).S_c(z_{k-1})\ldots))).$$

  The idea is that the term $\mathsf{s}^k(x)$, for $k>0$, is translated to an element of the numeric system $\langle\{\widehat{0},\widehat{1},\widehat{2},\ldots,\widehat{n},\ldots\},S_d(x)\rangle$, achieved by counting the elements from x up to $\widehat{k+x}$ using $S_d(x)$.
- The third is that we have not given the encoding of choice term $\sum_{i\in I}\varphi_i\lambda_i.T_i$. The reader can readily give it by himself/herself. It is simply a combination of the encodings for the prefix terms and the conditional terms. The translation of $\sum_{1\leq i\leq k}\varphi_i\lambda_i.T_i$ takes the following form

  $$(\widetilde{c})(\prod_{1\leq i\leq n_1} D_{c_i^1}(t_i^1)\,|\,\ldots\,|\,\prod_{1\leq i\leq n_k} D_{c_i^k}(t_i^k)\,|\,c_1^1(z_1^1)\ldots c_{n^k}^k(z_{n^k}^k).\sum_{1\leq i\leq k}[\![\varphi_i\lambda_i.T_i]\!]_{\mathsf{Th}}),$$

  where $\widetilde{c}=c_1^1\ldots c_{n^k}^k$.

It is easy to see that the translation is sound and complete with respect to the operational semantics in the sense of the next proposition.

Proposition 6. *Suppose P is a $\mathbb{VPC}$-process that does not contain any occurrences of $<$. The following correspondences hold:*
(i) If $P\stackrel{\lambda}{\longrightarrow}P'$ then $[\![P]\!]_{\mathsf{Th}}\rightarrow^*\stackrel{[\![\lambda]\!]_{\mathsf{Th}}}{\longrightarrow}P_1=_{\mathbb{VPC}_{\mathsf{Th}}}[\![P']\!]_{\mathsf{Th}}$ for some $\mathbb{VPC}_{\mathsf{Th}}$-process P_1;
(ii) If $[\![P]\!]_{\mathsf{Th}}\stackrel{\lambda}{\longrightarrow}P_1$ then $P\stackrel{\lambda'}{\longrightarrow}P'$ for some $\mathbb{VPC}$-process P' and some λ' such that $P_1=_{\mathbb{VPC}_{\mathsf{Th}}}[\![P']\!]_{\mathsf{Th}}$ and $[\![\lambda']\!]_{\mathsf{Th}}=\lambda$.

It is possible to strengthen Proposition 6. In fact the composition of the relation

$$\{(P,[\![P]\!]_{\mathsf{Th}})\mid P \text{ is a } \mathbb{VPC} \text{ process}\}$$

with $=_{\mathsf{Th}}$ is a subbisimilarity. A subbisimilarity is a generalization of the absolute equality from a binary relation on one model to a binary relation from one calculus to another. See [Fu12b] for details.

$$
\begin{aligned}
[\![\mathbf{0}]\!]_{\mathsf{Th}} &\stackrel{\text{def}}{=} \mathbf{0},\\
[\![\tau.T]\!]_{\mathsf{Th}} &\stackrel{\text{def}}{=} \tau.[\![T]\!]_{\mathsf{Th}},\\
[\![a(x).T]\!]_{\mathsf{Th}} &\stackrel{\text{def}}{=} a(x).[\![T]\!]_{\mathsf{Th}},\\
[\![\overline{a}(t).T]\!]_{\mathsf{Th}} &\stackrel{\text{def}}{=} \begin{cases} \overline{a}(\widehat{n}).[\![T]\!]_{\mathsf{Th}}, & \text{if } t \equiv \underline{n},\\ \overline{a}(x).[\![T]\!]_{\mathsf{Th}}, & \text{if } t \equiv x,\\ (c)(D_c(t) \mid c(z).\overline{a}(z).[\![T]\!]_{\mathsf{Th}}), & \text{if } t \equiv \mathsf{s}^l(x) \text{ for some } l > 0; \end{cases}\\
[\![S \mid T]\!]_{\mathsf{Th}} &\stackrel{\text{def}}{=} [\![S]\!]_{\mathsf{Th}} \mid [\![T]\!]_{\mathsf{Th}},\\
[\![(a)T]\!]_{\mathsf{Th}} &\stackrel{\text{def}}{=} (a)[\![T]\!]_{\mathsf{Th}},\\
[\![\mathit{if}\,\varphi\,\mathit{then}\,T]\!]_{\mathsf{Th}} &\stackrel{\text{def}}{=} (c_1 \ldots c_k)\left(\prod_{1\le i\le k} D_{c_i}(t_i) \mid c_1(z_1).\cdots c_k(z_k).\mathit{if}\,\varphi'\,\mathit{then}\,[\![T]\!]_{\mathsf{Th}}\right),\\
&\quad \text{where } \varphi' \equiv \varphi''\{\widehat{n_1}/\underline{n_1},\ldots,\widehat{n_j}/\underline{n_j}\},\ \varphi \equiv \varphi''\{t_1/z_1,\ldots,t_k/z_k\},\\
&\quad \underline{n_1},\ldots,\underline{n_j} \text{ are the numerals in } \varphi, \text{ and } t_1,\ldots,t_k \text{ are the terms}\\
&\quad \text{in } \varphi \text{ that are of the form } \mathsf{s}^l(x) \text{ for some } l > 0;\ \text{we may regard}\\
&\quad \mathit{if}\,\varphi'\,\mathit{then}\,[\![T]\!]_{\mathsf{Th}} \text{ as } [\![\mathit{if}\,\varphi\,\mathit{then}\,T]\!]_{\mathsf{Th}};\\
[\![!a(x).T]\!]_{\mathsf{Th}} &\stackrel{\text{def}}{=} !a(x).[\![T]\!]_{\mathsf{Th}},\\
[\![!\overline{a}(t).T]\!]_{\mathsf{Th}} &\stackrel{\text{def}}{=} \begin{cases} !\overline{a}(\widehat{n}).[\![T]\!]_{\mathsf{Th}}, & \text{if } t \equiv \underline{n},\\ !\overline{a}(x).[\![T]\!]_{\mathsf{Th}}, & \text{if } t \equiv x,\\ (c)(D_c(t) \mid c(z).!\overline{a}(z).[\![T]\!]_{\mathsf{Th}}), & \text{if } t \equiv \mathsf{s}^l(x) \text{ for some } l > 0. \end{cases}
\end{aligned}
$$

Fig. 7. Encoding of $\mathbb{VPC}$ into Turing Complete $\mathbb{VPC}_{\mathsf{Th}}$

Theorem 4. *Suppose P, Q are $\mathbb{VPC}$-processes that do not contain any occurrences of $<$. Then $P =_{\mathbb{VPC}} Q$ if and only if $[\![P]\!]_{\mathsf{Th}} =_{\mathbb{VPC}_{\mathsf{Th}}} [\![Q]\!]_{\mathsf{Th}}$.*

Proof. In view of Proposition 1, we only have to prove the theorem for the external bisimilarities. Let $\mathcal{R}$ be the following relation

$$
\left\{ ((\widetilde{c})(P_1 \mid \ldots \mid P_k), (\widetilde{c})(Q_1 \mid \ldots \mid Q_k)) \;\middle|\; \begin{array}{l} c \text{ is } c_1,\ldots,c_j \text{ for some } j;\ P_i \text{ is neither}\\ \text{a composition nor a localization; and}\\ [\![P_i]\!]_{\mathsf{Th}} \to^* Q_i, \text{ where } Q_i \text{ is obtained}\\ \text{from } [\![P_i]\!]_{\mathsf{Th}} \text{ by resolving the } D_c\text{'s.} \end{array} \right\}.
$$

Now $\mathcal{R}$ is a $\mathbb{VPC}_{\mathsf{Th}}$-bisimulation up to strong bisimilarity [Mil89a]. Notice that every $\mathbb{VPC}$-process is equal to a process of the form $(\widetilde{c})(P_1 \mid \ldots \mid P_k)$, where for each $i \in \{1,\ldots,k\}$, the process P_i is neither a composition nor a localization. □

So the translation $[\![\_]\!]_{\mathsf{Th}}$ is correct for the $\mathbb{VPC}$-processes that do not refer to the relation '$<$'. The restriction can be removed according to Proposition 5. We conclude that $\mathbb{VPC}$ is a submodel of every Turing complete $\mathbb{VPC}_{\mathsf{Th}}$. In other words it is the least expressive among all Turing complete value-passing calculi.

8 Conclusion

The present approach emphasizes that a value-passing calculus should be a self-contained model of computation and interaction. The formal treatment of the logic of the model comes with the decidability requirement. It has taken some time to reach this level of formality. The study of Milner [Mil89a] was conducted at an *ad hoc* manner. The semantics of his value-passing CCS is defined by translating the model into the pure CCS with arbitrary choice operator. Hoare's definition of his famous CSP [Hoa78, Hoa85], which is a value-passing calculus, is essentially algebraic. The operational semantics for CSP *a la* pure CCS is introduced in [Bro83, BHR84, Ros97]. The labeled transition semantics for the value-passing CCS appears in [HI93a, HI93b]. A more serious attempt to study the operational semantics of the value-passing calculi is reported in the seminal paper by Hennessy and Lin [HL95]. The significance of their work is that it points out the indispensable role the first order logic may play in the study of the semantics of the value-passing calculi. The present work can be seen as a further step that completes the picture outlined by Hennessy and Lin [HL95].

The observational equality of this paper is an application of the universal equality of Fu [Fu12b] to the value-passing calculi. This is an equality that differs from any equalities that have been proposed for the value-passing calculi. The algebraic theory of CSP has been extensively studied [BHR84, Ros97], with particular emphasis on the trace and failure semantics. In the literature on CSP, it is popular to see a set of algebraic laws as providing an axiomatic semantics. The algebraic theory of the value-passing calculi has been studied with motivation from the denotational semantics [Hen91, HI93a, HI93b]. The observational equivalence, the weak bisimilarity, and its symbolic characterization is systematically studied in [HL95]. Proof systems for these equivalences have also been studied in [Hen91, HI93a, HI93b, HL96]. These systems are parameterized over proof systems for the logics of data domains. The decidability of our equation system AS_{Th} compares favorably to these systems. The algorithmic aspect of the equivalence checking for the value-passing calculi is elaborated in [Lin93, Lin96, Lin98]. Our treatment to the value-passing calculi may cast new light on the equivalence checking algorithms for the value-passing processes. A survey of the symbolic approach is given by Ingolfsdottir and Lin [IL01].

Our formalization of the value-passing calculi makes it possible to carry out a refined study on the expressiveness of these models. There have been early efforts that attack the expressiveness issue. See for example [Pal03, FL10]. However it is fair to say that none of the results obtained so far is conclusive. In this paper we have studied the absolute expressiveness of the value-passing calculi by characterizing the Turing complete value-passing calculi in terms of the numeric systems. We have also studied the relative expressiveness of the value-passing calculi. It is shown in this paper that $\mathbb{VPC}$ is the least expressive value-passing calculus. The minimality result sheds new light on the value-passing calculi studied by previous researchers, all of those models being informal variants of $\mathbb{VPC}$. It is worth remarking that $\mathbb{VPC}$ is strictly less expressive than the π-calculus [Fu12b] and is strictly more expressive than the Interactive Machine Model [Fu12b].

We have emphasized the importance of confining our attention to the decidable fragment of the first order logic. The tradeoff is that $\simeq^s_{\mathsf{Th}}$ is much stronger than $=_{\mathsf{Th}}$. A challenging task is to prove or disprove that $\simeq^s_{\mathsf{Th}}$ and $=_{\mathsf{Th}}$ coincide on the finite control $\mathbb{VPC}_{\mathsf{Th}}$-terms. But a more urgent problem is the following.

Problem 1. Does $\simeq^s_{\mathbb{VPC}}$ coincide with $=_{\mathbb{VPC}}$ on the finite $\mathbb{VPC}$-terms?

The symbolic bisimilarity studied in this paper is the simplest of its kind. It is too strong for the infinite state processes. Consider for example the process $A_0 \stackrel{\text{def}}{=} a(x).\mathit{if}\ x$ is even $\mathit{then}\ \overline{b}(x) + a(x).\mathit{if}\ x$ is odd $\mathit{then}\ \overline{b}(x)$ and the process $A_1 \stackrel{\text{def}}{=} a(x).\overline{b}(x) + a(x).\mathit{if}\, x$ is even $\mathit{then}\, \overline{b}(x) + a(x).\mathit{if}\, x$ is odd $\mathit{then}\, \overline{b}(x)$. An implementation of $\mathit{if}\, x$ is even $\mathit{then}\, \overline{b}(x)$ is $(c)(\overline{c}(\underline{0}) \mid !c(y).\mathit{if}\, x = y\ \mathit{then}\, \overline{b}(x)\ \mathit{else}\, \overline{c}(\mathsf{s}(\mathsf{s}(y))))$. The term $\mathit{if}\, x$ is odd $\mathit{then}\, \overline{b}(x)$ is defined similarly. It is clear that $A_0 =_{\mathbb{VPC}} A_1$. On the other hand $A_0 \not\simeq^s_{\mathbb{VPC}} A_1$. The transition $A_1 \stackrel{a(x)}{\longrightarrow}_\top \overline{b}(x)$ can be simulated by A_0. There is however no boolean PA-partition on $\{x\}$ that witnesses the simulation. If we relax on the boolean restriction, then intuitively the set $\{\exists z.x = 2*z, \exists z.x = 2*z+1\}$ forms a 'partition'. In order to carry out investigation along this line, the symbolic approach must be modified in a couple of ways. Firstly the proper power of the Peano system should be retained. Specifically the addition operator '$+$' and the multiplication operator '$*$' are necessary to produce much more powerful specifications. Secondly the symbolic bisimilarity should be defined by a family of relations indexed by the first order formulas [HL95, IL01]. The introduction of the arithmetic operators does not change the grammar of $\mathbb{VPC}$. No $\mathbb{VPC}$-terms would contain any arithmetic operators. So the logic expressions of $\mathbb{VPC}$ are still decidable. The extra expressive power is only exploited in verification. We may ask the following question.

Problem 2. What is the symbolic theory that exploits the richer Peano theory?

The equation system AS_{Th} provides an effective method to check the symbolic bisimilarity of two finite $\mathbb{VPC}_{\mathsf{Th}}$-terms. A natural question to ask is how to extend AS_{Th} to a complete system for the finite control $\mathbb{VPC}_{\mathsf{Th}}$-terms. Hennessy, Lin and Rathke have discussed the issue in [HL97, Rat97a, Rat97b, HLR97]. It should be routine to adapt their approach and Milner's original approach [Mil89b] to $\mathbb{VPC}_{\mathsf{Th}}$. Additional care should be taken to divergence [LDH02, LDH05, Fu12a]. The details are yet to be worked out.

There are other aspects of the value-passing calculi that are worth investigating. One could for example take a look at the box equality introduced by Fu and Zhu [FZ11]. One could also take a look at the algorithm complexities for a number of decidability problems. The general methodology proposed in this paper has laid down a firm foundation for the solutions to these problems.

Acknowledgement. The author would like to thank the members of BASICS, especially to Yijia Chen, Huan Long and Jianxin Xue, for their interest and feedbacks. The support from NSFC (60873034, 61033002) and STCSM (11XD1402800) is gratefully acknowledged.

References

[BHR84] Brookes, S., Hoare, C., Roscoe, A.: A theory of communicating sequential processes. Journal of ACM 31, 560–599 (1984)

[Bro83] Brookes, S.: A Model of Communicating Sequential Processes. PhD thesis, Oxford University (1983)

[Cut80] Cutland, N.: Computability: An Introduction to Recursive Function Theory. Cambridge University Press (1980)

[DNMV90] De Nicola, R., Mantanari, U., Vaandrager, F.: Back and forth bisimulations. In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458, pp. 152–165. Springer, Heidelberg (1990)

[FL10] Fu, Y., Lu, H.: On the expressiveness of interaction. Theoretical Computer Science 411, 1387–1451 (2010)

[FR74] Fischer, M., Rabin, M.: Super-exponential complexity of presburger arithmetic. In: Karp, R. (ed.) Complexity of Computation, pp. 27–41. American Mathematical Society (1974)

[Fu12a] Fu, Y.: Nondeterministic structure of computation (2012)

[Fu12b] Fu, Y.: Theory of interaction (2012)

[FZ11] Fu, Y., Zhu, H.: The name-passing calculus (2011)

[Gö31] Gödel, K.: Über formal unentscheidbare sätze der principia mathematica und verwandter systeme. Monatshefte für Mathematik und verwandter Systeme I 38, 173–198 (1931)

[Hen91] Hennessy, M.: A proof system for communicating processes with value-passing. Journal of Formal Aspects of Computer Science 3, 346–366 (1991)

[HI93a] Hennessy, M., Ingólfsdóttir, A.: Communicating processes with value-passing and assignment. Journal of Formal Aspects of Computing 5, 432–466 (1993)

[HI93b] Hennessy, M., Ingólfsdóttir, A.: A theory of communicating processes with value-passing. Information and Computation 107, 202–236 (1993)

[HL95] Hennessy, M., Lin, H.: Symbolic bisimulations. Theoretical Computer Science 138, 353–369 (1995)

[HL96] Hennessy, M., Lin, H.: Proof systems for message passing process algebras. Formal Aspects of Computing 8, 379–407 (1996)

[HL97] Hennessy, M., Lin, H.: Unique fixpoint induction for message-passing process calculi. In: Proc. Computing: Australian Theory Symposium (CAT 1997), vol. 8, pp. 122–131 (1997)

[HLR97] Hennessy, M., Lin, H., Rathke, J.: Unique fixpoint induction for message-passing process calculi. Science of Computer Programming 41, 241–275 (1997)

[HM85] Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. Journal of ACM 32, 137–161 (1985)

[Hoa78] Hoare, C.: Communicating sequential processes. Communications of ACM 21, 666–677 (1978)

[Hoa85] Hoare, C.: Communicating Sequential Processes. Prentice Hall (1985)

[IL01] Ingólfsdóttir, A., Lin, H.: A symbolic approach to value-passing processes. In: Bergstra, J., Ponse, A., Smolka, S. (eds.) Handbook of Process Algebra, pp. 427–478. North-Holland (2001)

[LDH02] Lohrey, M., D'Argenio, P.R., Hermanns, H.: Axiomatising divergence. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 585–596. Springer, Heidelberg (2002)

[LDH05] Lohrey, M., D'Argenio, P., Hermanns, H.: Axiomatising divergence. Information and Computatio 203, 115–144 (2005)

[Lin93] Lin, H.: A verification tool for value-passing processes. In: Proceedings of 13th International Symposium on Protocol Specification, Testing and Verification, IFIP Transactions (1993)

[Lin96] Lin, H.: Symbolic transition graphs with assignment. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 50–65. Springer, Heidelberg (1996)

[Lin98] Lin, H.: "On-the-fly" instantiation of value-passing processes. In: Proc. FORTH/PSTV 1998 (1998)

[Mil89a] Milner, R.: Communication and Concurrency. Prentice Hall (1989)

[Mil89b] Milner, R.: A complete axiomatization system for observational congruence of finite state behaviours. Information and Computation 81, 227–247 (1989)

[MPW92] Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes. Information and Computation 100, 1–40 (Part I), 41–77 (Part II) (1992)

[Opp78] Oppen, D.: A $2^{2^{2^{pn}}}$ upper bound on the complexity of presburger arithmetic. Journal of Computer and System Sciences 16, 323–332 (1978)

[Pal03] Palamidessi, C.: Comparing the expressive power of the synchronous and the asynchronous π-calculus. Mathematical Structures in Computer Science 13, 685–719 (2003)

[Par81] Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981)

[Pre29] Presburger, M.: Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In: Warsaw Mathematics Congress, vol. 395, pp. 92–101 (1929)

[Pri78] Priese, L.: On the concept of simulation in asynchronous, concurrent systems. Progress in Cybernatics and Systems Research 7, 85–92 (1978)

[PS95] Parrow, J., Sangiorgi, D.: Algebraic theories for name-passing calculi. Information and Computation 120, 174–197 (1995)

[Rat97a] Rathke, J.: Symbolic Techniques for Value-Passing Calculi. PhD thesis, University of Sussex (1997)

[Rat97b] Rathke, J.: Unique fixpoint induction for value-passing processes. In: Proc. LICS 1997. IEEE Press (1997)

[Rog87] Rogers, H.: Theory of Recursive Functions and Effective Computability. MIT Press (1987)

[Ros97] Roscoe, A.: The Theory and Practice of Concurrency. Prentice Hall (1997)

[San93] Sangiorgi, D.: From π-calculus to higher order π-calculus–and back. In: Gaudel, M.-C., Jouannaud, J.-P. (eds.)TAPSOFT 1993. LNCS, vol. 668, pp. 151–166. Springer, Heidelberg (1993)

[Tho89] Thomsen, B.: A calculus of higher order communicating systems. In: Proc. POPL 1989, pp. 143–154 (1989)

[Tho93] Thomsen, B.: Plain chocs — a second generation calculus for higher order processes. Acta Informatica 30, 1–59 (1993)

[Tho95] Thomsen, B.: A theory of higher order communicating systems. Information and Computation 116, 38–57 (1995)

[vGW89] van Glabbeek, R., Weijland, W.: Branching time and abstraction in bisimulation semantics. In: Information Processing 1989, pp. 613–618. North-Holland (1989)

Proving Safety of Traffic Manoeuvres on Country Roads*

Martin Hilscher, Sven Linker, and Ernst-Rüdiger Olderog

Department of Computing Science, University of Oldenburg, Germany
{hilscher,linker,olderog}@informatik.uni-oldenburg.de

Abstract. We adapt the Multi-lane Spatial Logic MLSL, introduced in [1] for proving the safety (collision freedom) of traffic manoeuvres on multi-lane motorways, where all cars drive in one direction, to the setting of country roads with two-way traffic. To this end, we need suitably refined sensor functions and length measurement in MLSL. Our main contribution is to show that also here we can separate the purely spatial reasoning from the underlying car dynamics in the safety proof.

1 Introduction

The safety of road traffic can be increased by new assistance systems that are based on suitable sensors and communications between cars. Reasoning about car safety (collision freedom) is reasoning about hybrid systems involving car dynamics and spatial considerations. To simplify this reasoning, we proposed in [1] to separate the purely spatial reasoning from the car dynamics. To formalise this idea, we introduced a dedicated *Multi-lane Spatial Logic* (MLSL) for expressing spatial properties on multi-lane motorways, where the traffic is flowing on several lanes, but in one direction. We focused on the lane-change manoeuvre.

MLSL is inspired by Moszkowski's interval temporal logic [2], Zhou, Hoare and Ravn's Duration Calculus [3], and Schäfer's Shape Calculus [4]. MLSL is a two-dimensional extension of interval temporal logic, where one dimension has a continuous space (the position in each lane) and the other has a discrete space (the number of the lane). In MLSL we can, for example, express that a car has reserved a certain space on its lane. Safety then amounts to proving that under certain assumptions the reservation of different cars are always disjoint.

In this paper we turn to the more intricate setting of (multi-lane) country roads, with two-way traffic. We investigate the safety of an overtaking manoeuvre in the presence of opposing traffic. The original definition of MLSL only allowed for qualtitative reasoning, which is not aedequate for the definition of the overtaking protocol. Whenever a car changes into opposing traffic, it has to check beforehand whether the free space is sufficiently large for completing the manoeuvre. Hence we extend MLSL with the possibility to measure lengths.

* This research was partially supported by the German Research Council (DFG) in the Transregional Collaborative Research Center SFB/TR 14 AVACS.

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 196–212, 2013.

Our findings are that modulo these small extensions, the setting of MLSL is well suited to cover this setting and manoeuvre.

Related Work. Safety on multi-lane motorways was investigated extensively in the context of the California PATH project, where manoeuvres of car platoons including lane change have been studied [5]. Here the car dynamics was an integral part of the safety reasoning. Safety in urban traffic scenarios has been studied in [6]. The manoeuvres include lane change, double lane change with opposing traffic, right and left turns. The analysis is based on an abstract graph representing two possibly conflicting car trajectories and on some simplifying assumptions like a constant speed of the cars involved.

The idea of *separating* the dynamics from the control layer is pursued in the controller design for hybrid systems. In [7] the authors introduce abstraction and refinement in a hierarchical design of hybrid control systems. In [8] the synthesis of control laws for piecewise-affine hybrid systems is introduced. In [9] controller patterns were proposed to simplify safety proofs for two cooperating traffic agents. Proving system correctness across a number of different abstraction layers continues a line pioneered in the ProCoS (Provably Correct Systems) project [10], in which He Jifeng made significant contributions.

The novelty of our approach is that the control layer is given by spatial properties formalised in MLSL. In this paper we extend this approach to country roads with two-way traffic. To this end, we refine the abstract traffic model of [1] by taking traffic directions into account, by defining new sensor functions, and by adding length measurement in the logic MLSL.

2 Abstract Model

Throughout this paper we work with an abstract model of (multi-lane) country roads with two-way traffic that emphasises spatial properties, but hides the car dynamic as much as possible. In this model a country road has an infinite extension with positions represented by the real numbers, and lanes are represented by natural numbers $0, 1, \ldots, n$. At each moment of time each car, with a unique identity denoted by letters A, B, ..., has its position *pos*, speed *spd*, and acceleration *acc*. On country roads cars can move in two directions, one with increasing position values, in pictures shown from left to right, and one with decreasing position values, in pictures shown from right to left.

The abstract model is introduced by allowing for each car only local views of this traffic. A view of a car E comprises a contiguous subset of lanes, and has a bounded extension. A view containing all lanes with an extension up to a given constant, the *horizon*, is called *standard view*.

What a car "knows" of its view is expressed by formulas in the multi-lane spatial logic MLSL introduced in [1]. It extends interval temporal logic [2] to two dimensions, one with a continuous space (the position in each lane) and the other with a discrete space (the number of the lane). Such a formula consists of a finite list of lanes, where each lane is characterized by a finite sequence of

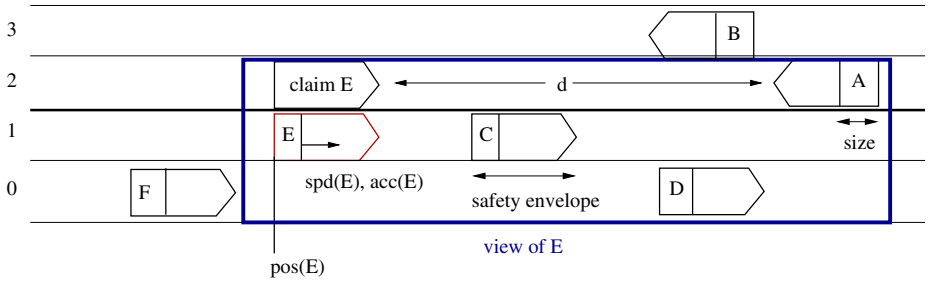


Fig. 1. View of car E covering a bounded extension of lanes 0, 1, and 2. Car E sees its own reservation and claim for preparing a change from lane 1 into the opposing traffic of car A on lane 2 to overtake car C ahead in lane 1. In this view, E does not see the cars B and F.

segments. A segment is either occupied by a car, say E, or it is empty (*free*). For instance, in the view of car E shown in Fig. 1, the following formula ϕ holds:

$$\phi \equiv \left\langle \begin{array}{c} \mathit{free} \frown \mathit{cl}(E) \frown (\mathit{free})^d \frown A \\ \mathit{free} \frown \mathit{re}(E) \frown \mathit{free} \frown C \frown \mathit{free} \end{array} \right\rangle,$$

where $\frown$ is the *chop operator* of interval temporal logic; it serves to separate adjacent segments in a lane. In the logic we can distinguish whether a car E has *reserved* a space in a lane ($\mathit{re}(E)$) or only *claimed* a space ($\mathit{cl}(E)$) for a planned lane change manoeuvre. We stipulate that reserved and claimed spaces have the extension of the *safety envelopes* of the cars, which include at each moment the speed dependent braking distances. The formula ϕ expresses also that there is a distance d between the claim of car E and the opposing car A.

We shall use such formulas as guards and location invariants of abstract controllers for car manoeuvres. In a technical realisation of such controllers, the properties that may appear in the formulae stipulate suitable sensors of the cars, for instance distance sensors.

2.1 Traffic Snapshot

We recall definitions from [1], extending them where needed. Let $\mathbb{L} = \{0, \ldots, N\}$, for some fixed $N \geq 1$, denote the set of lanes, with typical elements l, m, n. We assume a globally unique identifier for each car and take $\mathbb{I}$ as the set of all such *car identifiers*, with typical elements $A, B, \ldots$. To formalise two-way traffic, we refine the setting of [1] in two ways. First, we assume a *border* $b \in \mathbb{L}$ such that traffic on the lanes $l \leq b$ normally drives in the direction of increasing values of $\mathbb{R}$ (from left to right), and traffic on the lanes $l > b$ normally drives in the direction of decreasing values of $\mathbb{R}$ (from right to left). Only on the lanes b and $b+1$ cars may temporarily drive in opposite direction to perform an overtaking manoeuvre. Second, we partition $\mathbb{I}$ into the sets $\mathbb{I}_{\rightarrow}$ and $\mathbb{I}_{\leftarrow}$, i.e., $\mathbb{I}_{\rightarrow} \cup \mathbb{I}_{\leftarrow} = \mathbb{I}$ and $\mathbb{I}_{\rightarrow} \cap \mathbb{I}_{\leftarrow} = \emptyset$. The subscripts indicate the *driving direction* of the cars. Cars in

$\mathbb{I}_{\rightarrow}$ drive from left to right, and cars in $\mathbb{I}_{\leftarrow}$ drive from right to left. For simplicity we assume $\mathbb{I}_{\rightarrow}$ and $\mathbb{I}_{\leftarrow}$ to be countably infinite.

Definition 1 (Traffic snapshot). *A* traffic snapshot $\mathcal{TS}$ *is a structure* $\mathcal{TS} = (res, clm, pos, spd, acc)$, *where* res, clm, pos, spd, acc *are functions*

- $res : \mathbb{I} \to \mathcal{P}(\mathbb{L})$ *such that* $res(C)$ *is the set of lanes that car* C *reserves,*
- $clm : \mathbb{I} \to \mathcal{P}(\mathbb{L})$ *such that* $clm(C)$ *is the set of lanes that car* C *claims,*
- $pos : \mathbb{I} \to \mathbb{R}$ *such that* $pos(C)$ *is the position of the rear of car* C *on its lane,*
- $spd : \mathbb{I} \to \mathbb{R}$ *such that* $spd(C)$ *is the current speed of car* C*,*
- $acc : \mathbb{I} \to \mathbb{R}$ *such that* $acc(C)$ *is the current acceleration of car* C*.*

Let $\mathbb{TS}$ *denote the set of all traffic snapshots.*

Definition 2 (Transitions). *The following* transitions *describe the changes that may occur at a traffic snapshot* $\mathcal{TS} = (res, clm, pos, spd, acc)$*. We use the overriding notation* $\oplus$ *of Z for function updates [11].*

$$\begin{aligned}
\mathcal{TS} \xrightarrow{t} \mathcal{TS}' \quad \Leftrightarrow \quad & \mathcal{TS}' = (res, clm, pos', spd', acc) \\
& \wedge \forall C \in \mathbb{I}\colon pos'(C) = pos(C) + spd(C) \cdot t + \tfrac{1}{2} acc(C) \cdot t^2 \\
& \wedge \forall C \in \mathbb{I}\colon spd'(C) = spd(C) + acc(C) \cdot t \qquad (1) \\
\mathcal{TS} \xrightarrow{\mathsf{acc}(C,a)} \mathcal{TS}' \quad \Leftrightarrow \quad & \mathcal{TS}' = (res, clm, pos, spd, acc') \\
& \wedge acc' = acc \oplus \{C \mapsto a\} \qquad (2) \\
\mathcal{TS} \xrightarrow{\mathsf{c}(C,n)} \mathcal{TS}' \quad \Leftrightarrow \quad & \mathcal{TS}' = (res, clm', pos, spd, acc) \\
& \wedge |clm(C)| = 0 \wedge |res(C)| = 1 \\
& \wedge \{n+1, n-1\} \cap res(C) \neq \emptyset \\
& \wedge clm' = clm \oplus \{C \mapsto \{n\}\} \qquad (3) \\
\mathcal{TS} \xrightarrow{\mathsf{wd\ c}(C)} \mathcal{TS}' \quad \Leftrightarrow \quad & \mathcal{TS}' = (res, clm', pos, spd, acc) \\
& \wedge clm' = clm \oplus \{C \mapsto \emptyset\} \qquad (4) \\
\mathcal{TS} \xrightarrow{\mathsf{r}(C)} \mathcal{TS}' \quad \Leftrightarrow \quad & \mathcal{TS}' = (res', clm', pos, spd, acc) \\
& \wedge clm' = clm \oplus \{C \mapsto \emptyset\} \\
& \wedge res' = res \oplus \{C \mapsto res(C) \cup clm(C)\} \qquad (5) \\
\mathcal{TS} \xrightarrow{\mathsf{wd\ r}(C,n)} \mathcal{TS}' \quad \Leftrightarrow \quad & \mathcal{TS}' = (res', clm, pos, spd, acc) \\
& \wedge res' = res \oplus \{C \mapsto \{n\}\} \\
& \wedge n \in res(C) \wedge |res(C)| = 2 \qquad (6)
\end{aligned}$$

The transitions allow for the passage of time (1), with cars moving along the road, and a change of acceleration (2). These two transitions model abstractly the dynamic aspects of cars. To prepare for a lane change, a car may *claim* a neighbouring lane, which can be thought of as setting the turn signal, while it is not already in progress of changing lanes or has set the turn signal already (3). A car may *reserve* a previously claimed lane (5). Also, it may withdraw claims (4) and reservations (6), as long as at least one lane remains reserved by a car.

2.2 View

In our safety proof we will restrict ourselves to finite parts of a traffic snapshot $\mathcal{TS}$ called views, the intuition being that the safety of manoeuvres can be shown using local information only.

Definition 3 (View). *A* view V *is defined as a structure* $V = (L, X, E)$*, where*

- $L = [l, n] \subseteq \mathbb{L}$ *is an interval of lanes that are visible in the view,*
- $X = [r, t] \subseteq \mathbb{R}$ *is the extension that is visible in the view,*
- $E \in \mathbb{I}$ *is the identifier of the car under consideration.*

A subview *of* V *is obtained by restricting the lanes and extension we observe. For this we use sub- and superscript notation:* $V^{L'} = (L', X, E)$ *and* $V_{X'} = (L, X', E)$*, where* L' *and* X' *are subintervals of* L *and* X*, respectively.*

For a car E *and a traffic snapshot* $\mathcal{TS} = (res, clm, pos, spd, acc)$ *we define the* standard view *of* E *as*

$$V_s(E, \mathcal{TS}) = (\mathbb{L}, [pos(E) - h, pos(E) + h], E),$$

where the horizon h *is chosen such that a car driving at maximum speed can, with lowest deceleration, come to a standstill within the horizon* h *plus twice the distance it maximally takes to perform the overtake, i.e., the worst distance needed to pass a car and for changing lanes into the opposing traffic and back again. This way a car can be perceived early enough when planning an overtaking manoeuvre.*

We can give a rough estimate for the horizon h. Let us assume that a car E driving at 100 km/h wants to overtake a slower car travelling at 80 km/h. Passing the slower car will then take about 17 seconds. In this time the car travels approximatly 500 m. Furthermore, we assume an overapproximation for the braking distance of 50 m for driving at 100 km/h and 40 m for 80 km/h. Then we can safely overapproximate h with 1.5 kilometers, i.e., twice the travelling distance plus $500m$. Thus h exceeds both braking distances plus the distance needed for two lane changes.

Sensor Function. In [1] we introduced a sensor function $\Omega_E : \mathbb{I} \times \mathbb{TS} \to \mathbb{R}_+$ for each car E which, given a car identifier C and a traffic snapshot, provides the length of the car C, as perceived by E. For country roads we need to change this definition taking the opposing driving directions of cars into account.

Definition 4 (New sensor function). *We define the new sensor function* $\Omega_E^{new} : \mathbb{I} \times \mathbb{TS} \to \mathbb{R}$ *as:*

$$\Omega_E^{new}(C, \mathcal{TS}) = \begin{cases} \Omega_E(C, \mathcal{TS}) \text{ for } C \in \mathbb{I}_{\rightarrow} \\ -\Omega_E(C, \mathcal{TS}) \text{ for } C \in \mathbb{I}_{\leftarrow} \end{cases}$$

For a view $V = (L, X, E)$ and a traffic snapshot $\mathcal{TS} = (res, clm, pos, spd, acc)$ we introduce the following abbreviations:

$$res_V : \mathbb{I} \to \mathcal{P}(L) \text{ with } res_V(C) = res(C) \cap L \tag{7}$$

$$clm_V : \mathbb{I} \to \mathcal{P}(L) \text{ with } clm_V(C) = clm(C) \cap L \tag{8}$$

$$len_V : \mathbb{I} \to \mathcal{P}(X) \text{ with } len_V(C) = \begin{array}{l}[\min(pos(C), pos(C) + \Omega_E^{new}(C, \mathcal{TS})),\\ \max(pos(C), pos(C) + \Omega_E^{new}(C, \mathcal{TS}))] \cap X\end{array} \tag{9}$$

The functions (7) and (8) are restrictions of their counterparts in $\mathcal{TS}$ to the sets of lanes considered in this view. The function (9) gives us the part of the road that car E perceives as occupied by a car C, cut at the edges of the view's extension. We changed the corresponding definitions of [1]. The new definition of len_V is due to the opposing traffic. The changes of res_V and clm_V correct a technical mistake in the original paper. We note that the results from [1] remain valid even under the modified model given above.

2.3 Multi-Lane Spatial Logic with Length Measurement

We define the *multi-lane spatial logic* MLSL extended by length measurement for road segments. We start from two kinds of variables. The set of *car variables* ranging over car identificators is denoted by CVar, with typical elements c, d. To refer to the car owning the current view, we use a special variable ego $\in$ CVar. The set of *real variables* ranging over real numbers is RVar, with CVar$\cap$RVar $= \emptyset$ and typical elements x, y. The set of all *variables* is Var $=$ CVar $\cup$ RVar, with typical element u, v. For the length measurement we need real-valued terms.

Definition 5 (Terms). *Real-valued* terms θ *of MLSL are given by the syntax*

$$\theta ::= r \mid x \mid f(c_1, \ldots, c_n) \mid g(\theta_1, \ldots, \theta_n),$$

where $r \in \mathbb{R}$, $x \in$ RVar, $c_1, \ldots, c_n \in$ CVar, *and* f, g *are* n*-ary function symbols with* $\mathbb{R}$ *as result type. We denote the set of all terms with* Θ.

Formulae of MLSL are built up from six atoms, boolean connectors and first-order quantification. Furthermore, we use two *chop* operations. The first chop is denoted by $\frown$ like for interval logics, while the second chop operation is given only by the vertical arrangement of formulae. Intuitively, a formula $\phi_1 \frown \phi_2$ is satisfied by a view V with the extension $[r, t]$, if V can be divided at a point s into two subviews V_1 and V_2, where V_1 has the extension $[r, s]$ and satisfies ϕ_1 and V_2 has the extension $[s, t]$ and satisfies ϕ_2, respectively. A formula $\begin{smallmatrix}\phi_2\\ \phi_1\end{smallmatrix}$ is satisfied by V with the lanes l to n, if V can be split along a lane m into two subviews, V_1 with the lanes l to m and V_2 with the lanes $m + 1$ to n, where V_i satisfies ϕ_i for $i = 1, 2$.

Definition 6 (Syntax). *Formulae ϕ of MLSL with length measurement are given by the syntax*

$$\phi ::= true \mid u = v \mid \ell = \theta \mid free \mid re(\gamma) \mid cl(\gamma)$$
$$\mid \phi_1 \wedge \phi_2 \mid \neg\phi_1 \mid \exists v\colon \phi_1 \mid \phi_1 \frown \phi_2 \mid \begin{matrix}\phi_2\\ \phi_1\end{matrix}$$

where γ is a variable or a car identifier, u and v are variables, and θ is a term. We denote the set of all MLSL formulae by Φ.

In a length measurement $\ell = \theta$ the letter ℓ stands for *length* as in [3], We use ϕ^θ as an abbreviation for the formula $\phi \wedge \ell = \theta$.

Definition 7 (Valuation and Modification). *A* valuation *is a function* $\nu\colon \mathrm{Var} \to \mathbb{I} \cup \mathbb{R}$ *that* respect types *(maps car variables to car identifiers and real variables to real values). Inductively we lift ν to a function $val_{\mathcal{TS},\nu}\colon \Theta \to \mathbb{I} \cup \mathbb{R}$:*

$$val_{\mathcal{TS},\nu}(\theta) = \begin{cases} r & \text{if } \theta = r \in \mathbb{R} \\ \nu(x) & \text{if } \theta = x \in \mathbb{R} \\ \hat{f}_{\mathcal{TS}}(\nu(c_1), \ldots, \nu(c_n)) & \text{if } \theta = f(c_1, \ldots, c_n) \\ \hat{g}(val_{\mathcal{TS},\nu}(\theta_1), \ldots, val_{\mathcal{TS},\nu}(\theta_n)) & \text{if } \theta = g(\theta_1, \ldots, \theta_n), \end{cases}$$

where $\hat{f}_{\mathcal{TS}}, \hat{g}$ are the interpretations of f, g, with subscript $\mathcal{TS}$ indicating a possible dependency on a traffic snapshot $\mathcal{TS}$. For a valuation ν we use the overriding notation $\nu \oplus \{v \mapsto \alpha\}$ to denote the modified valuation*, where the value of v is modified to α. We assume that this modification respects types.*

We define partitioning of discrete intervals. We need this notion to have a clearly defined chopping operation, even on the empty set of lanes.

Definition 8 (Chopping discrete intervals). *Let I be a discrete interval, i.e., $I = [l, n]$ for some $l, n \in \mathbb{L}$ or $I = \emptyset$. Then $I = I_1 \ominus I_2$ if and only if $I_1 \cup I_2 = I$, $I_1 \cap I_2 = \emptyset$, and both I_1 and I_2 are discrete intervals such that $\max(I_1) + 1 = \min(I_2)$, or $I_1 = \emptyset$ or $I_2 = \emptyset$ holds.*

Since the semantics of formulae depends on both views and valuations, we will only consider valuations ν which are *consistent* with the current view $V = (L, X, E)$, which means that we require $\nu(\mathrm{ego}) = E$. In the following definition, we require that the spatial atoms hold only on a view with exactly one lane and an extension greater than zero. In the semantics of *free*, we abstract from cars visible only at the endpoints of the view.

Definition 9 (Semantics). *The* satisfaction $\models$ *of formulae with respect to a traffic snapshot $\mathcal{TS}$, a view $V = (L, X, E)$ with $L = [l, n]$ and $X = [r, t]$, and a valuation ν consistent with V is defined inductively as follows:*

$$\mathcal{TS}, V, \nu \models true \qquad \text{for all } \mathcal{TS}, V, \nu$$
$$\mathcal{TS}, V, \nu \models u = v \quad \Leftrightarrow \quad \nu(u) = \nu(v)$$

$$
\begin{array}{lcl}
\mathcal{TS}, V, \nu \models \ell = \theta & \Leftrightarrow & |X| = val_{\mathcal{TS},\nu}(\theta) \\
\mathcal{TS}, V, \nu \models \mathit{free} & \Leftrightarrow & |L| = 1 \ \mathit{and}\ |X| > 0 \ \mathit{and} \\
 & & \forall\, C \in \mathbb{I}\colon len_V(C) \cap (r,t) = \emptyset \\
\mathcal{TS}, V, \nu \models re(\gamma) & \Leftrightarrow & |L| = 1 \ \mathit{and}\ |X| > 0 \ \mathit{and}\ \nu(\gamma) \in \mathbb{I} \ \mathit{and} \\
 & & res_V(\nu(\gamma)) = L \ \mathit{and}\ X = len_V(\nu(\gamma)) \\
\mathcal{TS}, V, \nu \models cl(\gamma) & \Leftrightarrow & |L| = 1 \ \mathit{and}\ |X| > 0 \ \mathit{and}\ \nu(\gamma) \in \mathbb{I} \ \mathit{and} \\
 & & clm_V(\nu(\gamma)) = L \ \mathit{and}\ X = len_V(\nu(\gamma)) \\
\mathcal{TS}, V, \nu \models \phi_1 \wedge \phi_2 & \Leftrightarrow & \mathcal{TS}, V, \nu \models \phi_1 \ \mathit{and}\ \mathcal{TS}, V, \nu \models \phi_2 \\
\mathcal{TS}, V, \nu \models \neg\phi & \Leftrightarrow & \mathit{not}\ \mathcal{TS}, V, \nu \models \phi \\
\mathcal{TS}, V, \nu \models \exists v\colon \phi & \Leftrightarrow & \exists\, \alpha \in \mathbb{I} \cup \mathbb{R}\colon \mathcal{TS}, V, \nu \oplus \{v \mapsto \alpha\} \models \phi \\
\mathcal{TS}, V, \nu \models \phi_1 \frown \phi_2 & \Leftrightarrow & \exists\, s\colon r \le s \le t \ \mathit{and} \\
 & & \mathcal{TS}, V_{[r,s]}, \nu \models \phi_1 \ \mathit{and}\ \mathcal{TS}, V_{[s,t]}, \nu \models \phi_2 \\
\mathcal{TS}, V, \nu \models \begin{smallmatrix}\phi_2\\ \phi_1\end{smallmatrix} & \Leftrightarrow & \exists\, L_1, L_2\colon L = L_1 \ominus L_2 \ \mathit{and} \\
 & & \mathcal{TS}, V^{L_1}, \nu \models \phi_1 \ \mathit{and}\ \mathcal{TS}, V^{L_2}, \nu \models \phi_2
\end{array}
$$

We write $\mathcal{TS} \models \phi$ if $\mathcal{TS}, V, \nu \models \phi$ for all views V and consistent valuations ν.

We remark that both chop modalities are associative. For the definition of the controller we employ some abbreviations. In addition to the usual definitions of $\vee, \rightarrow, \leftrightarrow$ and $\forall$, we use a single variable or car identifier γ as an abbreviation for $re(\gamma) \vee cl(\gamma)$. Furthermore, we use the notation $\langle\phi\rangle$ for the two-dimensional modality *somewhere* ϕ, defined in terms of both chop operations:

$$\langle\phi\rangle \equiv \mathit{true} \frown \begin{pmatrix} \mathit{true} \\ \phi \\ \mathit{true} \end{pmatrix} \frown \mathit{true}.$$

In the following, the main application of the somewhere modality is to abstract the exact positions on the road from formulae, e.g., to identify overlaps of claims and safety envelopes. If a view V satisfies the formula $\exists c\colon \langle cl(\mathrm{ego}) \wedge re(c)\rangle$, then there is a part on some lane in V occupied by both the claim of the car under consideration and the safety envelope of some car c.

3 Controllers for Overtaking with Perfect Knowledge

We now present a protocol realised by several controllers for the overtaking manoeuvre. The complexity of the controllers depends on the knowledge available for each car about the surrounding cars. For simplicity, we assume here *perfect knowledge*, i.e., all cars know the extension of all safety envelopes within their view. This assumption can be formalised by instantiating the sensor function Ω_E (see Sec. 2.2) as $\Omega_E(I, \mathcal{TS}) = se(I, \mathcal{TS})$, where *se* is a function returning the *safety envelope*, an overapproximation of the braking distance. This implies that

the car E can perceive a car C entering its view as soon as part of the safety envelope enters E's view. Clearly, this is an idealization that in reality would require very powerful sensors for each car. For the setting of motorways we have considered also more realistic sensor functions in [1], but for country roads we leave this for future work.

We extend the lane-change automaton LCP of [1] to take the new situation at the border lanes into account. On the rest of the country road the automaton works as before, with slight modifications described in Sec. 3.3. We assume that ego $\in \mathbb{I}_{\rightarrow}$, i.e., ego is driving in direction of increasing values of $\mathbb{R}$. For the border lanes, we need to employ communication with the surrounding cars to guarantee safety, even though we assume perfect knowledge. Otherwise cars from the lanes next to the border b, i.e., from $b-1$ and $b+2$, could move into the free space either in front of the car C, which E wants to overtake, or into the lane that E needs for overtaking C. For example, in Fig. 1 car D could move into lane 1 and then block the space E needs to move back. Similarly, the car B would be allowed to move into lane 2 and block the space E needs to pass C. To prohibit this behaviour, we use a *helper automaton* as described in Sec. 3.2.

To simplify the safety proof in Sec. 4 we structure the overtaking manoeuvre into the following three phases, also shown in Fig. 2.

1. Change lanes into opposing traffic.
2. Pass the car driving in front.
3. Change lanes back into the original driving lane.

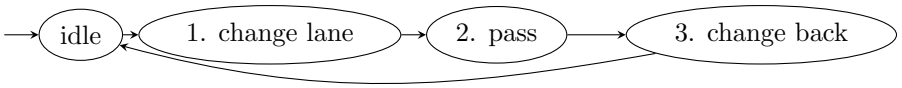


Fig. 2. Protocol for overtaking

For phase 1 and 3 we will present controllers as timed automata [12] with data variables ranging over $\mathbb{I}$ and $\mathbb{L}$. We allow for MLSL formulae as transition guards and invariants and evaluate them over the standard view V_s (see Sec. 2.2) of the car the controller is implemented in. Furthermore we use the labels of the transition system of $\mathcal{TS}$ as actions. Finally, communication is modelled via *broadcast messages*, similar to UPPAAL [13]. The notation we use for communication is inspired by CSP [14]. For phase 2 we will omit the presentation of an automaton due to its simplicity but give a detailed description.

Our controllers rely on a spatial decomposition of the overtaking manoeuvre. To formalise this, we assume function symbols interpreted as functions yielding certain distances for cars travelling at speeds determined by the current traffic snapshot $\mathcal{TS}$ (cf. Def. 7):

- $d(t)$ yields the maximal *distance* a car can travel within a given time t;
- $d_{lc}(E)$ yields the distance needed for a *lane change* of a car E driving at a speed determined by $\mathcal{TS}$;

- $d_{pass}(E, C)$ yields the distance needed for a car E to *pass* a car C in front, given their speeds determined by $\mathcal{TS}$;
- $d_{lcb}(E, C)$ yields the distance needed for a *lane change back* of a car E after passing car C with speeds determined by $\mathcal{TS}$;
- $d_{max} = d(t_{max})$, where the time t_{max} is large enough for a car to safely change lanes twice and pass another car.

3.1 Overtaking Protocol

Overall we want to maintain the property that all reservations are disjoint. We formalise the unwanted situations in MLSL by the formula *collision check*:

$$cc \equiv \exists c : c \neq \text{ego} \wedge \langle re(\text{ego}) \wedge re(c) \rangle \quad .$$

Now we will present controllers for the three phases of the manoeuvre that will maintain this property in a setting with opposing traffic. To construct the complete controller for the overtaking protocol we fuse the locations without any outgoing arcs, named q_4 in Fig. 4 and Fig. 5, with the initial locations of the next phase of the protocol.

Changing Lanes into Opposing Traffic. Intuitively, we can describe this phase of the manoeuvre as follows. The car ego drives on lane n (its *original lane*) and wants to change into the *target lane* $n + 1$ next to n. Then ego first sets a claim on the target lane and checks the following three properties:

1. Does the claim intersect with the reservation or claim of any other car?
2. Is there enough space on the original lane to change back during the completion of the manoeuvre?
3. Is there enough space for the overtaking manoeuvre on the target lane?

Should one of these conditions be violated, withdraws ego its claim and therefore aborts the manoeuvre. Otherwise, ego proceeds by sending a message that it starts an overtaking manoeuvre, turning the claim into a reservation and starting moving over to the target lane. Furthermore, this message obliges the overtaken car to keep its velocity constant.

We formalise the first property, the *potential collision*, i.e., car ego's claim intersects with another car's claim or reservation, in the following MLSL formula:

$$pc \equiv \exists c : c \neq \text{ego} \wedge \langle cl(\text{ego}) \wedge c \rangle \quad .$$

For checking that there is enough space on the target lane and on the original driving lane, we need two functions k_1 and k_2 (see Fig. 3):

$$k_1 : \mathbb{I} \times \mathbb{I} \to \mathbb{R}_+ \text{ with } (E, C) \mapsto d_{lc}(E) + d_{pass}(E, C)$$

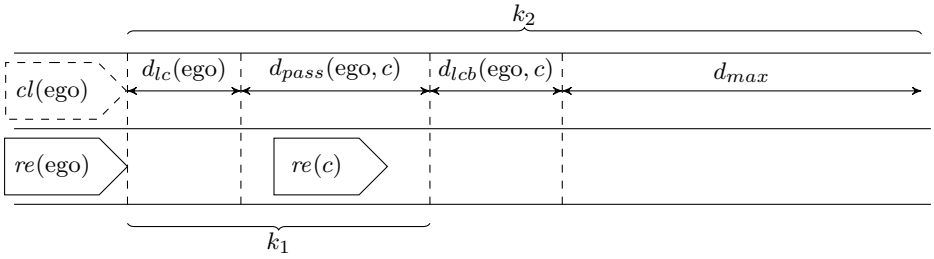


Fig. 3. Lengths referenced to during overtaking manoeuvre

is the distance needed by car E for a lane change with its current speed and for passing the car C, during which E may accelerate to a higher speed.

$$k_2 : \mathbb{I} \times \mathbb{I} \to \mathbb{R}_+ \text{ with } (E, C) \mapsto k_1(E, C) + d_{lcb}(E, C) + d_{max}$$

adds the distance it takes E to change back into the original lane (with the speed assumed while passing car C in front plus the maximal distance d_{max} a car on the opposing lane can travel in the time it takes to overtake C. With these two functions we can now formulate the remaining two properties.

$$esol(c) \equiv \left\langle re(\text{ego}) \frown (free \frown re(c) \frown free)^{k_1(\text{ego},c)} \frown free^{d_{lcb}(\text{ego},c)} \right\rangle$$

states that there is *enough space on the original lane*. Since ego is currently holding a claim, it can reserve only one lane, and hence *esol* refers to ego's original lane. In front of the reservation there has to be enough space such that ego can safely pass the car c and then fit in front of c (cf. Fig. 3).

$$estl(c) \equiv \left\langle cl(\text{ego}) \frown free^{k_2(\text{ego},c)} \right\rangle$$

states that there is *enough space on the target lane*. We fix the target lane by checking for ego's claim. In front of the claim there has to be enough space to complete the overtaking manoeuvre plus the maximal distance travelled by an opposing car during the time of the manoeuvre (cf. Fig. 3).

Figure 4 shows the controller part for this phase of the manoeuvre. We start in q_0 which we assume to be safe, i.e., satisfying $\neg cc$. We hold ego's original driving lane in the variable n and make use of the additional variable l saving the target lane. We use the constant t_{lc} for the time that is needed to change lanes. Note that due to the guard $n = b$ this controller only takes effect if ego is driving on the border and ego's target lane is the lane with opposing traffic.

Passing the Car Ahead. The controller for the car ego to pass by car c of the manoeuvre is rather simple. The car ego accelerates to a higher speed, remains at that speed until it has passed car c, and initiates the next phase of the manoeuvre. Furthermore, whenever ego receives a request for a lane change from a car c, i.e., the message $req?c$, ego checks whether c wants to occupy space needed for the overtaking manoeuvre, i.e., whether $\langle re(\text{ego}) \frown true \frown cl(c) \rangle$ holds.

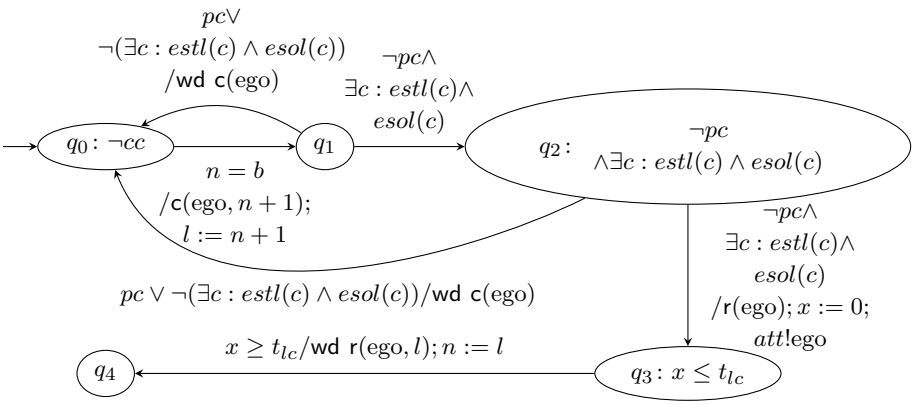


Fig. 4. Controller for the lane-change manoeuvre into opposing traffic

If this is the case, ego denies c's request by sending the message $no!c$. We omit the presentation of the described controller.

Changing Lanes Back into Original Driving Lane. The controller for changing the lane back into the original driving lane (see Fig. 5) is a drastically simplified version of the controller in Sec. 3.1. We already established with *esol* that there is enough space to change lanes back into the original lane. We only need to set the variable l to the original lane $n-1$, then we can claim it ($\mathsf{c}(\text{ego}, n-1)$), reserve the lane ($\mathsf{r}(\text{ego})$), and change over within t_{lc} time.

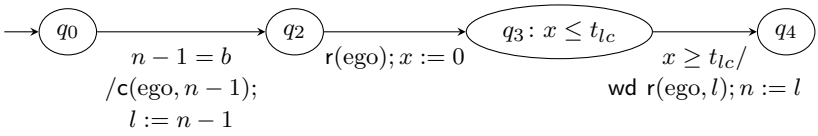


Fig. 5. Controller LCP for the lane-change manoeuvre with perfect knowledge back into orignal lane

3.2 Helper Controller

The automaton in Fig. 6 is glued to the helper controller from [1], to prohibit cars from moving to space used during the overtaking manoeuvre. As soon as a car starts to overtake another car C, it calls for *attention* by sending its identity on channel *att* (see Fig. 4). After receiving the message $att?c$, the car C can realise that it is being overtaken, by checking whether $\langle re(c) \frown free \frown re(\text{ego})\rangle$ holds. Then C reacts to each request $req?d$ of other cars to move into the free space in front of it with a negative reply $no!d$. The end of the overtaking manoeuvre can be perceived by C by means of the formula $\langle re(ego) \frown free \frown re(h)\rangle$. Combined with the lane change controller from [1] this maintains that no other car moves into the free space in front of the car which is being overtaken.

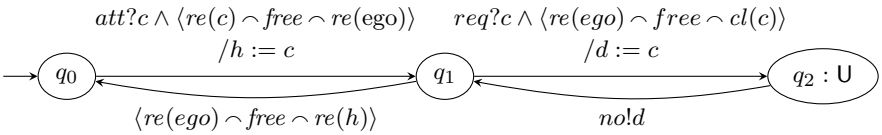


Fig. 6. Additional automaton for helper controller

3.3 Changing Lanes for Non-border Lane Manoeuvres

In Fig. 7 we present the slightly modifed controller from [1], which in cooperation with the previous controllers ensures safety of the complete manoeuvre. This controller is responsible for the lane change if the car is not moving into opposing traffic during the lane change.

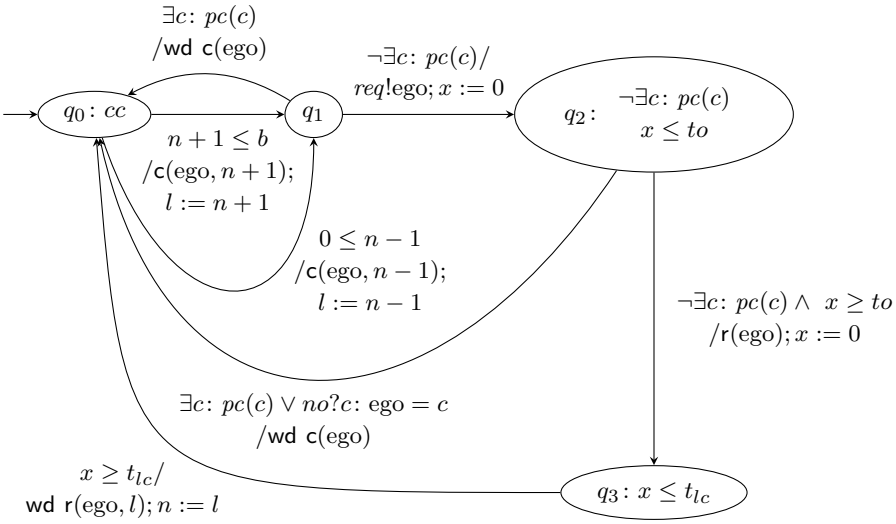


Fig. 7. Controller LCP for the lane-change manoeuvre with perfect knowledge on non-border lanes

Similar to the controller in Sec. 3.1 it maintains the current driving lane in the variable n and the target lane in the variable l. Upon attempting a lane change ego sets its claim on the target lane and brodcasts a request $req!$ego awaiting an answer for the next to time units. If no abort message $no?c : \text{ego} = c$ arrives within this time-out, ego turns its claim into a reservation and starts to move over. After t_{lc} the manoeuvre is completed and ego withdraws the original reservation. We assume the time-out to to be large enough such that a car which previously moved into opposing traffic can answer within this time bound.

Note that ego receives a *no* message in the following two situations:

1. The car C behind ego is currently being overtaken.
2. ego wants to change into a lane a car D is currently using to overtake.

In the first case C sends a *no* to ensure that the space ego perceives *free* is maintained *free* for the overtaking car behind C. In the second case D sends a *no* message to ego to maintain that the space it currently uses to complete the overtake manoeuvre remains *free*.

4 Safety of Overtaking Manoeuvre

In this section, we will give an informal safety proof of the controllers shown in Sec. 3. Since the reservations of the model are overapproximations of the braking distance of cars together with their physical size, we understand safety as the non-overlapping of any reservations. Claims however may overlap, since they represent only intentions of future car positions.

We make the following assumptions on the country road traffic. The assumptions are slight extensions to the assumptions for our proof of motorway situations [1], to take the opposing traffic into account.

Assumption A1. There is an *initial safe* traffic snapshot $\mathcal{TS}_0$.

Assumption A2. Every car C is equipped with a *distance controller* that keeps the safety property invariant with respect to all cars driving in the same direction as C under time and acceleration transitions, i.e., for every transition $\mathcal{TS} \xrightarrow{t} \mathcal{TS}'$ and $\mathcal{TS} \xrightarrow{\mathsf{acc}(C,a)} \mathcal{TS}'$ if $\mathcal{TS}$ is safe also $\mathcal{TS}'$ is safe.

Informally, this means that the distance controller admits a positive acceleration of C only if the space ahead permits this. Also, if the car ahead is slowing down, the distance controller has to initiate braking (with negative acceleration) of C to reduce the extension of its reservation (the safety envelope).

Assumption A3. Every car is equipped with a controller implementing the protocol in Sec. 3.1.

Finally, we assume the horizon h to be of appropriate size, as stated in Sec. 2.

Assumption A4. The horizon h is $h = se_{max} + 2 \cdot d_{max}$. This length stems from the distance we need for the overtaking manoeuvre to take place, as shown in the controllers, as well as the maximal length of the safety envelope se_{max}.

Theorem 1 (Safety). *Under the assumptions A1 to A4, the protocol specifying the overtaking procedure of Section 3 is safe.*

Proof. We sketch the safety proof by analysing the three phases of the protocol. Let $V = (\mathbb{L}, X, E)$ be the view of the car $E \in \mathbb{I}_{\rightarrow}$ performing the overtaking of a car C and $\mathcal{TS}_0$ the initial safe snapshot (A1).

The first phase is essentially the lane change procedure of our previous work [1], under different assumptions. However, the new assumptions are stronger in the following sense. First, we only allow for the lane change in one direction, i.e., if E is driving on lane b, it may only change to lane $b + 1$. Furthermore, in the proof for motorway situations, the horizon was at least of the length of the safety envelope. Since by A4 the horizon is still large enough, the lane change

manoeuvre can take place in a safe manner. In this phase, the controller has to ensure that in the third phase, there is still enough space in front of C to change back to lane b. Therefore, at the instant, when E extends its reservation, it broadcasts its ID on the channel *att*. Now C is the only car, where the guard in the helper automaton (Fig. 6) is satisfied. Any car attempting to change to b in front of C, i.e., where C would serve as a helper car, will receive the denial to change lanes, until E has changed back to b, as stated in the guard on the transition back to the initial location of the helper automaton. Hence, there will be still free space for E on lane b in the third phase of the protocol. Similarly, a car $D \in \mathbb{I}_{\leftarrow}$ trying to perform a lane change to lane $b+1$ will receive a denial directly from ego.

Now assume that phase one was successful, i.e., E started its lane change at snapshot $\mathcal{TS}_0$ and successfully changed to lane $b+1$ on a subsequent snapshot $\mathcal{TS}_1$. Then the free space in front of E on $b+1$ is still at least $d_{pass}(E,C) + d_{lcb}(E,C) + (d_{max} - d(t_{lc}))$. The car E needs $d_{pass}(E,C)$ space during this phase. Due to the definition of d_{max} an opposing car may cover at most a distance of $d_{max} - 2 \cdot d(t_{lc})$ in the passing phase. Hence in the worst case after this phase, there is still at least $d_{lcb}(E,C) + (d_{max} - d(t_{lc}) - (d_{max} - 2 \cdot d(t_{lc})) = d_{lcb}(E,C) + d(t_{lc})$ free space left. Similarly to the description of the first phase, E denies all cars wanting to change to its current lane the permission to do so.

For the last phase of the protocol, observe that $d_{lcb}(E,C)$ is the space needed by E to change back to lane b, while $d(t_{lc})$ is the largest distance an opposing car may cover when driving at maximal velocity. Hence after the final lane change, i.e., before E removes its reservation from lane $b+1$, in the worst case the start of the envelope of the opposing traffic is directly in front of E's reservation, but they are not yet overlapping. Since the removal of reservations is instantaneous, there is no point in time where the these reservations can overlap.

Now assume that a car $D \in \mathbb{I}_{\leftarrow}$ on lane $b+1$ with a view $V' = (\mathbb{L}, X', D)$ outside the horizon of E is also planning an overtaking manoeuvre of a car A. Observe that this also implies that E is outside the horizon of D, i.e., $\max(len_V(E)) < pos(D) - h$. An unsafe situation may only occur, when the spaces used to change back on the original lanes overlap, i.e., when

$$\begin{aligned}
& \max(len_V(E)) + d_{lc}(E) + d_{pass}(E,C) + d_{lcb}(E,C) \\
> \;& \min(len_{V'}(D)) - (d_{lc}(D) + d_{pass}(D,A) + d_{lcb}(D,A)) \\
\iff \;& \max(len_V(E)) + d_{lc}(E) + d_{pass}(E,C) + d_{lcb}(E,C) - d_{max} \\
> \;& \min(len_{V'}(D)) - (d_{lc}(D) + d_{pass}(D,A) + d_{lcb}(D,A)) - d_{max} \qquad (10)
\end{aligned}$$

Now by definition $d_{max} \geq d_{lc}(E) + d_{pass}(E,C) + d_{lcb}(E,C)$. Hence we get

$$\max(len_V(E)) \geq \max(len_V(E)) + \; d_{lc}(E) + d_{pass}(E,C) + d_{lcb}(E,C) - d_{max} \qquad (11)$$

By definition, we have $|len_{V'}(D)| \leq se_{max}$, and hence $\min(len_{V'}(D)) = pos(D) - |len_{V'}(D)| \geq pos(D) - se_{max}$. Since also $d_{lc}(D) + d_{pass}(D,A) + d_{lcb}(D,A) \leq d_{max}$, we have that

$$\begin{aligned}
&\min(len_{V'}(D)) - (d_{lc}(D) + d_{pass}(D, A) + d_{lcb}(D, A)) - d_{max} \\
&= pos(D) - |len_{V'}(D)| - (d_{lc}(D) + d_{pass}(D, A) + d_{lcb}(D, A)) - d_{max} \\
&\geq pos(D) - se_{max} - (d_{lc}(D) + d_{pass}(D, A) + d_{lcb}(D, A)) - d_{max}. \\
&\geq pos(D) - (se_{max} + 2 \cdot d_{max}) \qquad (12)
\end{aligned}$$

By putting (10), (11) and (12) together and using $h = se_{max} + 2 \cdot d_{max}$, we get $\max(len_V(E)) > pos(D) - h$, which means that E is visible within the standard view of D, and hence contradicts our assumption. □

5 Conclusion

The novelty in our approach is the identification of a level of abstraction that enables a purely spatial reasoning on safety. In [1] this was demonstrated for motorways. In this paper we showed that with small extensions to the previously developed setting, in particular explicit length measurement, we can also deal with the two-way traffic of country roads. We proved safety for arbitrarily many cars on country roads locally, by considering a limited amount of space.

In our future work, we would like to study variations of the assumptions made in our safety proofs. First of all, we will lift the assumption of perfect knowledge. In [1] we have already done this for the case of one-way motorway traffic. There the more realistic sensor function assumes that each car knows only the physical size of all other cars in its view; the safety envelope it knows only of itself. Further, we intend to study the scenarios of urban traffic as in [6]. Also, we plan to link our work to hybrid systems: a refinement of the spatial reasoning in this paper to the car dynamics is of interest. There we could benefit from the work of [7,8,9].

So far, our proof of the Safety Theorem 1 is by hand. We show that the transitions between traffic snapshots obeying our controllers preserve a suitable safety invariant. It is our aim to ultimately provide automatic support for such proofs. In [15] steps in this direction are presented. There a proof system for an extended version of the logic called EMLSL is introduced. EMLSL embraces temporal operators, so reasoning about the transitions can be conducted within this logic. However, mechanic support for the proofs remains a topic of future research.

References

1. Hilscher, M., Linker, S., Olderog, E.-R., Ravn, A.P.: An abstract model for proving safety of multi-lane traffic manoeuvres. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 404–419. Springer, Heidelberg (2011)
2. Moszkowski, B.: A temporal logic for multilevel reasoning about hardware. Computer 18, 10–19 (1985)

3. Zhou, C., Hoare, C., Ravn, A.: A calculus of durations. Information Processing Letters 40, 269–276 (1991)
4. Schäfer, A.: Axiomatisation and decidability of multi-dimensional duration calculus. Information and Computation 205, 25–64 (2007)
5. Lygeros, J., Godbole, D.N., Sastry, S.S.: Verified hybrid controllers for automated vehicles. IEEE Transactions on Automatic Control 43, 522–539 (1998)
6. Werling, M., Gindele, T., Jagszent, D., Gröll, L.: A robust algorithm for handling traffic in urban scenarios. In: Proc. IEEE Intelligent Vehicles Symposium, Eindhoven, The Netherlands, pp. 168–173 (2008)
7. Moor, T., Raisch, J., O'Young, S.: Discrete supervisory control of hybrid systems based on l-complete approximations. Discrete Event Dynamic Systems 12, 83–107 (2002)
8. Habets, L.C.G.J.M., Collins, P., van Schuppen, J.: Reachability and control synthesis for piecewise-affine hybrid systems on simplices. IEEE Transactions on Automatic Control 51, 938–948 (2006)
9. Damm, W., Hungar, H., Olderog, E.R.: Verification of cooperating traffic agents. International Journal of Control 79, 395–421 (2006)
10. He, J., et al.: Provably correct systems. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994. LNCS, vol. 863, pp. 288–335. Springer, Heidelberg (1994)
11. Woodcock, J., Davies, J.: Using Z – Specification, Refinement, and Proof. Prentice Hall (1996)
12. Alur, R., Dill, D.L.: A theory of timed automata. TCS 126, 183–235 (1994)
13. 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)
14. Hoare, C.A.R.: Communicating sequential processes. CACM 21, 666–677 (1978)
15. Linker, S., Hilscher, M.: Proof theory of a multi-lane spatial logic. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) ICTAC 2013. LNCS, vol. 8049, pp. 231–248. Springer, Heidelberg (2013)

Generic Models of the Laws of Programming

Tony Hoare

Microsoft Research, Cambridge, United Kingdom
thoare@microsoft.com

Abstract. The laws of programming are a collection of judgments about the equality and ordering of computer programs. A model of the laws is a mathematical description of the execution of programs, where the model has been proved to satisfy the laws. A generic model is one that has parameters that can be adjusted to the properties of a range of different programming languages and their differing implementations and differing applications. In this way, a generic model serves as the basis of a unifying theory of programming.

1 Introduction

The purpose of the laws of programming [8] is to specify and reason about the correctness of programs and of their implementations. The laws are also used directly to transform a program into an equivalent or better one, perhaps one that is more efficient in execution, or expressed in a useful normal form, or even expressed in a different language which obeys the same laws. The laws provide a common theoretical framework for the specification and use of a range of software engineering tools. Such tools include program analysers, generators, interpreters, compilers, optimisers and verifiers; also test case generators and tools for error detection, diagnosis and correction. It is obviously important that all of the tools provided should be based on the same laws of programming; otherwise there is a danger of mismatch across the interfaces between the tools.

The purpose of a mathematical model is to describe some aspect of physical reality, and give mathematical proof that it satisfies the laws which are used to reason about it. For the laws of programming, the reality described is the behaviour of a computer system while executing a program. The behaviour can be recorded as a trace of all actions performed inside and in the vicinity of a network of computers during program execution. The model specifies the correspondence between a program and the set of all its possible behaviours, in every possible environment of use. The model described in this essay is based firmly on the pomset model of [6].

The intricate details of the model clearly must depend on choice of a particular programming language, and on a particular notion and notation for program correctness. In some cases, it is desirable to extend the model to a larger language, or to simplify it for a particular special-purpose subset of the language. The subset may require a program to conform to a particular program design pattern,

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 213–226, 2013.

or to the use of a particular class library. Many successful tools, which have applied formal methods to practical programming problems, have only succeeded by exploiting a serendipitous combination of such particular circumstances.

The purpose of a generic model is to control and co-ordinate an inevitable proliferation of tools for software engineering. A generic model accepts parameters whose variations define the details of a wide range of more specialised models. For example, the model described in this essay allows variation in the definitions of the basic actions of the programming language, the primitive predicates of the specification language, and the basic types of object provided. It is even possible to vary the definitions of sequential and concurrent composition, and of the behaviour of computer memory. The resulting generic model is proved to satisfy the full set of laws of programming, no matter what the choice of parameters. Thus it is always valid to use in combination any set of tools that are based on the same laws.

Our strategy is to model a program as the set of traces of all its possible behaviours, in all possible circumstances of its use. Each trace is modelled as the set of all atomic actions that have occurred as a result of execution of the program. Let A be the set of all possible actions that need to be considered: it is the basic parameter of the model. The laws and models are developed in three stages, each of which builds upon its predecessor: (1) a model of traces, in which the carrier set is the set of all subsets of A, plus the element $\perp$ standing for undefined; it defines on traces the basic partial operations of sequential and concurrent composition. (2) a model of programs in which the carrier set is the powerset of all traces. It adds set union to its operators to represent choice (either non-deterministic, or controlled by condition, and a least fixed point operator to provide iteration/recursion to the programming language. (3) a model of program specifications (or other descriptions of program behaviour). It adds conjunction and a greatest fixed point to the operators, and provides Galois adjoints for the two programming operators. This essay concentrates on (1); for a fuller treatment of (2) and (3), see [8].

2 The Laws

The laws of programming constitute a mathematical structure with several algebraic components. The basic component is a partial order, $(C, \sqsubseteq, \perp)$. In the trace model, $\perp$ represents an undefined element, and $\sqsubseteq$ is Scott's refinement ordering. For programs, the ordering $\sqsubseteq$ is refinement: it holds between a program and one that is more determinate than it. $\perp$ is a program which has no executions, perhaps because the compiler has found a serious programming error in it. Between specifications, the relation $\sqsubseteq$ means logical implication. Between a program and a specification it means correctness. The familiar concept of refinement covers all these cases.

The next algebraic component is a monoid $(C, |, 1)$. The operator $|$ represents concurrent composition of traces. Applied to programs, it defines a program, all of whose traces are the concurrent composition of a trace of its first operand

with a trace of its second operand. Applied to specifications, it is a description of the concurrent composition of two programs, each of which is described one of the two specifications. The concurrency monoid is combined with the partial order in an ordered monoid, which requires also that | be monotonic wrto the ordering $\sqsubseteq$.

The next component is another ordered monoid, with the same ordering as before. The operator | is replaced by ; (standing for sequential composition). Furthermore, | and ; interact through an exchange law

$$(p \mid q)\ ;\ (p' \mid q')\ \sqsubseteq\ (q\ ;\ q') \mid (p\ ;\ p')\ .$$

Because ; and | share the same unit, four simpler corollaries emerge

$$(1)\ \ p\ ;\ (q \mid r)\ \sqsubseteq\ (p\ ;\ q) \mid r \qquad (2)\ \ (p \mid q)\ ;\ r\ \sqsubseteq\ p \mid (q\ ;\ r)$$

$$(3)\ \ p\ ;\ q\ \sqsubseteq\ p \mid q \qquad (4)\ \ \mid \text{commutes}$$

The final step is to add lattice operations $\cup$ and $\cap$ to the partial order. The $\cup$ operator represents choice between programs: only one of its operands is executed. It represents simple disjunction between specifications. The $\cap$ operator represents conjunction of program specifications. It provides the simplest and most useful way of combining many requirements for programs that have not yet been written. The operators | and ; are required to distribute through $\cup$, but not through $\cap$. In the model the operators also distribute through unions of arbitrary sets of operands.

3 The Trace Model

In the trace model, the carrier set is the set of all traces, where a trace is a subset of the set A of all actions; to this is adjoined a distinct undefined element $\perp$. The lattice structure is the discrete semilattice as defined by Scott, which has $\perp$ as its bottom, and all distinct elements of A are incomparable. The relational judgement $p \sqsubseteq q$ means that p is equal either to q or to $\perp$.

(1) Concurrency. The first and simplest operator to define is the parallel composition operator |.

$$p \mid q\ \ =\ p \cup q, \qquad \text{if } p \text{ and } q \text{ differ from } \perp.$$

$$=\ \perp, \qquad \text{otherwise}$$

This says that the trace of concurrent composition of two commands, when defined, consists exactly of the all the actions of each of its operands. Nothing is added, and nothing is taken away. The unit 1 of this operator is clearly the empty set of actions. This is an extremely weak definition of an operator: in fact, all other operators will be defined as special cases of it. There is no constraint whatever on the sequencing or simultaneity of the individual actions of the union. It even allows the CCS and CSP [4] style of concurrency, in which a successful

communication is a simultaneous occurrence of an action from p with an action from q, provided that one of them is an input and the other is an output of the same message on the same channel.

To give a generic model of concurrency, we introduce a parameter P, and use it to define the domain of the concurrency operator. P is assumed to be a symmetric relation between actions. In order for the generic concurrency operator to be defined, it is necessary that every action in the first operand p is related by P to every action in the second operand q.

$$p \mid' q \;=\; p \cup q \qquad \text{if } p \times q \text{ is contained in } P$$

Here, $\times$ is Cartesian product of sets. For brevity, the line which explicitly defines the undefined case is omitted. The associativity of this operator depends only on the fact that Cartesian product distributes through union. It does not depend on any particular properties of the parameter P. However, symmetry of the $\mid$ operator depends on the symmetry of P.

(2) Sequential Composition. To define sequential composition of traces, we introduce another parameter S to restrict its domain to a subset of P. For example, S may be a relation between actions expressing a notion of dependency or causality, which prevents one action being performed before another. A strong but simple definition of sequential composition states that none of the actions of the second operand q can occur before any of the actions of the first operand p:

$$p \;;\; q \;=\; p \mid' q, \qquad \text{if } (p \times q) \text{ is contained in } S.$$

This definition is strong because it requires that all actions of p should be completed before q begins. Note that it is not required that S be transitive.

Satisfaction of the exchange law between ; and $\mid$ depends solely on the form of the definition of sequential composition. It depends in no way on properties of the parameter S.

(3) Programs and specifications. The carrier set for the model of programs is the powerset of all subsets of subsets of A. The undefined element $\perp$ of the trace model is simply omitted.

Composition of commands, both sequential and concurrent, is defined by 'lifting to sets'. Following a common mathematical practice, if the formula for the typical member of a set is undefined, it is excluded from the set. For example, the clause '$(p \;;\; q) \neq \perp$' is omitted from the set-builder in the definition

$$pp \;;\; qq \;=\; \{p \;;\; q \mid p \text{ in } pp \text{ and } q \text{ in } qq\}$$

Distribution through union follows directly from the form of the above definition. Preservation of the other algebraic properties of the trace model follows from the following observation. Every algebraic equation and inequality in the statement of the laws has exactly two occurrences of each of its variables, one on each side of the equality or inequality symbol.

The new operator $\cup$ on programs is just set union. Distribution of $|$ and ; through union follows directly from the lifting equation shown above.

The carrier set for specifications is the same as that for programs. The new operation $\cap$ on specifications is just intersection of sets. Consequently, a program satisfies the conjunction of two specifications just if it satisfies each of the specifications separately. There are no new interaction laws which relate intersection to ; or $|$. This reflects the fact that there is no way of implementing conjunction of requirements in a modular way, by implementing programs for each component separately.

4 Programming Language Semantics

Our laws are strongly related to three of the traditional methods of defining the semantics of programming languages, (1) denotational semantics, (2) separation logic, and (3) operational semantics. The relationship is easy to express: the structural rules of (2) and (3) can be simply derived from the laws that are satisfied by (1) [19,20]. In the other direction, many of the laws can be derived both from (2) and from (3), particularly the last and least obvious of them, the exchange law. This gives encouraging evidence that the laws are faithful to a widely held understanding of the meaning of programs. It also gives evidence of the usefulness of the laws for all the applications for which (2) and (3) have been used in the past.

(1) The model itself is a denotational semantics, because it maps each program to a well-known mathematical structure, namely a set of traces, which are themselves defined as sets of actions. Reasoning about programs can now be conducted in terms of the model, using only pure mathematical reasoning. The reasoning is greatly assisted by the algebraic laws.

To simplify mathematical reasoning, early presentations of denotational semantics abstracted as far as possible away from the details of implementation of the language. For example, Scott represented deterministic sequential programs as partial functions, mapping the initial state in which the program is started onto the final state in which it ends; and everything else which happens during the process of execution is omitted.

Our goal is rather different from Scott's. It is to prove that the laws are valid for all possible abstractions from the execution of the program, including no abstraction at all. As a result, the model can be used directly to support program testing. A tool for program testing must make accessible to the programmer the entire set of actions that occurred on a test run of a program. Preferably this should take a form that explains the causes and consequences of each action, and so assist in the discovery of what went wrong and when.

(2) The purpose of a deductive semantics (originally called an axiomatic semantics by [7]) is to provide a valid pattern of proof that there are no errors in a program. The derivation of the structural proof rules for Hoare logic from

the laws of programming proceeds as follows. First, the primitive judgement of Hoare logic is defined in algebraic notation:

$$p\ \{q\}\ r \quad = \quad p\ ;\ q\ \sqsubseteq\ r$$

This enables the Hoare rule of consequence to be derived from associativity and monotonicity of ;. The rule for sequential composition is similarly derived. The rule for non-determinism follows from the distribution of ; through $\cup$. The modular concurrency rule of concurrent separation logic [10] is derived from the exchange law, and his frame rule is derived from its first corollary. Surprisingly, this direction of derivation can be reversed: the exchange law can be proved directly from O'Hearn's rule.

(3) The purpose of an operational semantics [11] is to show an abstract method of executing a program step by step, which generates just a single trace of it at a time. It has been widely used to guide the design of language implementations by interpreters or compilers. One of the original goals of an operational semantics was to discover a minimal set of concepts, operators and principles for programming. All other concepts are then shown to be implementable in terms of these primitives. Our goal is rather different from this. It is to give a reasonably simple and well-structured model, together with a set of parameters that enable it to be adapted directly to a wide range of likely variations. Thus there is no need to prove the correctness of the implementation of each variation.

The derivation of an operational semantics from the laws of programming starts by defining Milner's transition [12], in very much the same way as the Hoare triple:

$$p \xrightarrow{q} r \quad = \quad q\ ;\ r\ \sqsubseteq\ p$$

This is interpreted as saying that one of the ways of executing p is to execute q first and then continue with r. As a result, the CCS rule for prefixing is a tautology: $(p; q) \xrightarrow{p} q$. It is suggestive also to use a traditional notation for the 'silent' transition: $p \to r = r \sqsubseteq p$. Then the obvious rules of an operational semantics can be derived for ;. The exchange law gives a general (big-step) version of one of Milner's rules for concurrency in CCS, the one which describes a successful communication by synchronised input and output between concurrent processes. The first corollary of the exchange law gives the other concurrency law, which describes the effect of an unmatched input or output. Again the last two derivations can be reversed, giving a further support for the exchange law.

The associativity and distributivity laws for ; can be derived from the combination of the Hoare rules and the Milner rules for ; and $\cup$. However, there are many laws of programming that cannot be derived from either of these traditional forms of semantics. The commutation of | is an example. The traditional semantics are further weakened by placing syntactic restrictions on the parameters of the basic judgements. For example, the p and the r of the Hoare judgement must be assertions, and the q of the Milner transition must be an atomic action.

Our more general algebraic laws can be re-established by introducing an equivalence relation between programs. This can be defined by so-called structural equivalence laws, by contextual equivalence, by bisimulation, or by a combination of these. An equivalence relation provides an excellent way of raising the level of abstraction of a model; and it also provides additional laws. However, in this essay, we can get all the laws we want directly form a concrete model, without using an equivalence.

5 Variations

Introduction of the parameters P and S allows easy definition of several interesting variants of the operators of the trace model. They satisfy the same laws, and their satisfaction can be proved by the same proofs. In this section, we introduce a single new parameter D, standing for a dependency relation between actions. In terms of D we define a range of different values for S and P, giving different versions of the operators.

The concept of dependency is quite prevalent in Computer Science, as in other branches of Science. If x and y are actions in the same trace, the dependency xDy means that y could not have happened before x, or equivalently that x could not have happened after y. In a diagram of a trace, the actions are often drawn as points or circles or boxes, and the dependency xDy is drawn an arrow from the point for x to the point for y. In a Message Sequence Chart, events in the same service or thread are connected by downward-pointing arrows, and communications between services are drawn as horizontal arrows. In a hardware wave-form diagram, the events are the rising and falling edges of the voltage on a wire. Actions occurring on the same wire are connected by a horizontal line, with the direction of the arrow understood to be from left to right. Dependencies between events are represented by sloping arrows between actions on different lines.

1. $p \mid' q \;=\; p \cup q$, if $p \times q$ is contained in the inequality relation between actions.

This is nothing but the disjoint union of sets p and q. It means that no action of the combined trace requires simultaneous participation of both components of the concurrent combination. This is a common feature of most programming languages and of many process algebras–but not CCS or CSP, where input and output of the same message are regarded as the same atomic event.

2. $p \parallel q \;=\; p \mid q$, if $p \times q$ is contained in the negation of $(D^* \cap converse(D^*))$

This version of concurrency is undefined if there is a mutual dependency cycle that contains actions of both the two operands. In implementation, such a cycle would manifest itself as a deadlock between the concurrent components, which would prevent the combined trace from completing. The above definition prevents such a trace from even starting.

3. $p \mid\mid\mid q \;=\; p \mid q$, if $p \times q$ is contained in the negation of $(D^* \cup converse(D^*))$

This definition requires there to be no dependency at all between any actions of p and any actions of q. In CSP, $|||$ is the interleaving operator, which uses the same notation. With this operator deadlock is impossible, and so is interference (a.k.a. race condition) between the operands.

4. $p\ ;\ q\ =\ p \mid q$, if $p \times q$ is contained in the negation of the converse of D^*

The stronger version of sequential composition defined earlier uses D^* in place of S. This weaker definition is the one actually implemented by the compilers and hardware instruction pipelines on most modern computers. The weakening allows some of the actions of q to occur before some of the actions of p. But this only happens when the overtaking p-action does not depend on the overtaken q-action. As a consequence, the stronger strictly sequential definition still gives a trace that satisfies the weaker definition given above.

5. The above definitions still satisfy all the laws when $\mid$ on the right hand side is replaced by $\mid'$ or by $\|$ or by $|||$. (Unfortunately, this last case is vacuous. The resulting ; certainly satisfies the exchange law with $|||$, but only because these two operators are equal. Effectively, no interaction is possible between any components of a program).

6. Let us introduce another parameter H to the model. It is a set of traces that are considered Healthy, in the sense that they are free of certain generic errors which can afflict a program. Typical errors are division by zero, null pointer dereference, a false assertion, a race condition, etc. Let $(p; q)$ in H imply that p in H and q in H. This reflects the fact that a sequential composition is healthy only if both its components are healthy. Define a new operator to deliver only healthy results:

$$(p\ ;'\ q)\ =\ (p\ ;\ q) \qquad \text{if } (p\ ;\ q) \text{ in } H$$

This is the range-restriction of ; with respect to H. It preserves the algebraic properties of ;.

7. Some errors occurring in a trace can be attributed not as a fault in the program, but rather in some other program in its environment. Typically, the error is violation by the other program of a contract between the two programs, for example a false precondition or other assumption. Let us introduce another parameter F to denote the set of all traces containing a symptom of a failure which is actually the fault of its environment. Let $(p \mid q)$ in F imply that p is in F and that q is in F. Then we define

$$(p \mid' q)\ =\ (p \mid q) \qquad \text{if } (p \mid q) \text{ in } F.$$

Restricting the model to traces in F will now satisfy all the laws. The restriction embodies the assumption that the environment satisfies all its contracts.

The distinction between F and H defines the roles of particular components of a software engineering toolset, and of the form of semantic presentation on which they are based. For example, a program verifier and its deductive semantics

should check by proof that all traces in F are also in H. A test case generator should generate only traces in F that are not in H, with no false positives.

6 Objects and Behavioural Types

The actions provided by a conventional programming language usually inspect or update various objects that have been originally declared or otherwise allocated by the program. An object may be a simple variable of some built-in type, or a structured variable (eg., an array or structure), built from many simple variables. Concurrent languages often provide a range of different types of object, with different behaviours. Examples are: threads, semaphores, communication channels and messages sent on them. An object-oriented language provides class declarations, which enable the programmer to define the behaviour of new types of object.

Each action in a trace of a program may involve simultaneous participation of one or more objects. For example, an output action involves the outputting thread, the communication channel, and the message itself. For each trace and for each object, we identify the subset of actions of the trace which have involved that object. It is called the slice (or projection) of the object within the trace. We define the behaviour of a type (or class) of object like that of a program, as the set of all possible slices of that object in the traces of all possible programs which use an object of that type. We define an object as atomic if all its possible slices are totally ordered by D^*. Thus the first action of every object is its allocation, and each subsequent action has a unique D-predecessor. Similarly the last action of the object is its disposal, and all previous actions have a unique D-successor. Let Obj be the set of all atomic objects. We take this as an additional parameter of our model.

We illustrate the possible content of the Obj parameter by some examples. They are taken from [16], where they are diagrammed by simple graphs. Each example is an informal description of the behaviour of a particular class of concurrent object: semaphores, threads, communication channels, and variables. In some cases, variants are given.

(1) A thread is an atomic object, whose slice is a non-deterministic linearization of a trace generated by execution of the sequential program that runs on the thread. Except for the main thread, the first action involving a thread is a fork, which is a simultaneous action of its 'parent' thread. The last action is often a join, which is also shared by the parent or by another thread.

(2) The behaviour of a binary semaphore is a sequence of actions. The first action is a P. Subsequent executions of P alternate with executions of V, and the last action is a V.

(3) The behaviour of a channel is a structure consisting of the behaviour of atomic objects of three different types: an output port, an input port, and a set of messages. The behaviour of an output port is a sequence of output actions, and an input port is a sequence of input actions. A message engages in only two

actions. Its first action is participation in one of the outputs on the channel, and its last action is participation in one of the inputs. If no further constraints are placed on the messages, the above account describes the behaviour of a highly unreliable postal service, which can lose or duplicate or reorder messages. A more reliable service can be specified by adding constraints on the behaviour of the channel, as in the following variants.

(3a) A reliable fully-buffered channel is one in which for each output action (say the nth) there is a single message starting there and ending at the nth input. As a result, all output messages are delivered exactly once, and in the right order.

(3b) In a single-buffered channel, the life of the nth message is extended: its third (and last) action is the $(n + 1)$st output (or disposal of the port itself). This ensures that every read of the nth output happens before the $(n + 1)$st output, so a single buffer is sufficient to implement it.

(4) The behaviour of a variable (in strong memory) consists of a sequence of assignments and a set of messages carrying the assigned value to a read of it. Each assignment may be the start of many messages, or of none. Each message starts with an assignment (say the nth), and this is followed by a read of the value assigned. The third and final action of the message is the $(n + 1)$st assignment (or the disposal of the variable). The third action ensures that each assignment happens after all reads of the previous assignment, even if they occur in a different thread.

(4a) In a weak memory, this third action of each message is missing. Any necessary synchronisations must be performed explicitly by the user program, using fences.

7 Locality

The simple models of object behaviour given in the previous section permit arbitrary sharing of objects between the components of a concurrent program, each of which can at any time perform any one of the available updates upon it. However, if all objects are shared, it is impossible to write useful programs. For this reason, the hardware and intimate firmware of modern processors provide separate registers local to each thread, and they cannot be updated by any other thread. This section shows how the trace model can describe the general concept of locality, as applied to objects allocated by program, and not just to a fixed set of hardware registers.

The problem posed to the programmer by shared objects is called interference: the non-deterministic occurrence of interference from another thread leads to a race condition. The phenomenon is modelled as follows. Let x, y, and z be actions occurring (in that order) in the slice of an object l. Let t be a trace which contains x and z, but not y. Interference arises because y may be an arbitrary action occurring absolutely anywhere in the environment of t. This makes it impossible to reason about t in a modular fashion, that is in terms only

of those actions that occur inside t itself. We define the object l to be local to t if the phenomenon described above does **not** occur.

To introduce locality into our model, we split the dependency parameter D into two parts: $D = L+R$, where L stands for local dependency (between actions typically within the same thread), and R stands for remote dependency (typically between actions in different threads). We define an object as purely local if all its slices are totally ordered by L^*, and volatile if they are all totally ordered by R^*. But many objects have both kinds of dependence, with phases in which they are local to one trace, separated by actions which can transfer ownership. For example, consider a binary exclusion semaphore. The dependency of a P action on the immediately preceding V action is remote, because the V and the immediately following P can occur in different threads. However, if a thread contains a critical region surrounded by P and V, it is known that only the same thread can perform the next V operation on the semaphore. So this dependency of V on P is local.

It is important to emphasise that L is a parameter of the model, and it is often necessary to define the parameter in accordance with the ownership strategy adopted by a particular program. For example, a program may specify that ownership of an output port should be changeable after every third output, and an input port after every input of a prime number. This flexibility makes it impossible to give a complete and definitive semantics for concurrent programming. Fortunately, validity of the laws of programming is independent of the choice of the parameter L.

The formal definition of locality to a trace is the negation of a formal statement of the phenomenon which it is desired to avoid

l is local to t iff for all x, y, z in l, if xL^*yL^*z & x, z are in t, then y is in t.

Note that if $(l \cap t)$ is empty, then this definition is trivially satisfied, and also if $(l \cap t)$ contains only one action. Furthermore, if an object is local to $(p\ ;\ q)$ then it is also local to p and to q. This enables us to use the technique of variation 6 of the previous section to define a stronger version $;'$ of our earlier sequential composition:

$$p\ ;'\ q\ =\ p\ ;\ q, \qquad \text{if for all } l \text{ in } Obj,\ l \text{ is local to } (p\ ;\ q).$$

Volatile objects are local to every trace, so for them the effect of sequential composition is unchanged. For a purely local object, the new definition ensures that the last action in $(p \cap l)$ is related by L to the first action in $(q \cap l)$, whenever they both exist. This is what enables an assignment to a local variable to communicate the assigned value directly to the next assignment to that same variable in the same sequential piece of code. It is also what validates the simple laws of assignment like

$$v := 5\ ;\ v := 3xv \quad = \quad v := 5\ ;\ v := 15 \qquad \text{if } v \text{ is a local variable.}$$

8 Related Work

This essay explains and summarises results that were obtained in collaboration with many colleagues named as co-authors in the references. Some ideas of the last section may be so far unpublished.

9 Tribute

This essay honours my old friend and colleague He Jifeng. He first joined the Programming Research Group in Oxford in the mid-1980s, as a visitor sponsored by the Chinese Academy of Sciences, of which he is now a distinguished Member. He had a desk in an office shared with Jeff Sanders, a visitor from Australia. We published our first joint articles in 1986.

At that time, Jifeng spoke English with an accent which we found hard to understand. As a result, we thought that he too would find our English and Australian accents hard to interpret. From the first, we used to engage him in frequent conversations about our research. Whenever we explained something that we thought intricate, he would immediately respond gently with a soft 'Yes'. Naturally, we thought he must have misinterpreted our accents, so we explained the same idea again more slowly. Again he instantly responded 'Yes'. It did not take us long to realise that even the first 'Yes' meant that he had understood perfectly, not only our accents, but also the full intricacies of what we were trying to say.

In a short time, he had read the draft of my book 'Communicating Sequential Processes'. In it, I had claimed that nearly all the laws of the deterministic model of CSP were preserved by the non-deterministic model. At some stage he plucked up courage to send me a long list of these laws which were in fact invalid in the non-deterministic model. As a result, I deleted the claim from my book draft. He was always extremely polite, and never directly pointed out the many later mistakes that I made in my manuscript drafts of joint articles. However, his type-set version of the article quietly omitted or corrected each mistake.

Early in my career as an academic Computer Scientist, I read Tarski's article on the Calculus of Relations. I very much admired its style and content. Jifeng told me that just before coming to Oxford he had conducted a study group reading the same article, and had reached the same appreciation of it as me. But his group came to the reluctant conclusion that Tarski's ideas had no application in Computer Science. Little did we know what was to come!

In 1987, Jifeng and I decided that we should direct our future research towards the goal of unifying theories of programming. Our motive was to bring order and structure into the profusion of previously published theories, each of which was claimed by its author to be a successful rival to all the others. It was rare for any new theory to be further developed by others, or even by its author. We hoped that a unifying theory would be generic, so that a wide range of existing theories could be shown to be special cases. We hoped that it would provide a framework permitting further collaborative development among researchers, both on theory and on practical implementation.

At the basis of our unifying theory of sequential programming, we took Tarski's relational model and algebra. Everything worked well until we came to Chapter 8 on Concurrency. I wrote and rewrote that chapter around ten times, and none of the versions had anything like the elegance of Tarski's work. Jifeng checked the mathematics and uncomplainingly typeset each version, — in those days he used Postscript, not Latex. Eventually he reported that the length of the latest draft chapter stood at twenty pages, and that he had typed all two hundred of them. I took the hint that it was time to move on; we then dealt with process algebras, including CSP. Our book was eventually published in 1998 [9]. It never achieved a wide circulation, and is now freely available on the web.

I now believe that the basic mistake that I made was in 1969. At that time, I was inspired by the axiomatic method as used by algebraists to define concepts like groups and rings. The first half of [7] used algebraic axioms to define computer arithmetic in a way that permitted various treatments of overflow. It would have been natural to use the same kind of axioms to define the semantics of a programming language. But unthinkingly I chose to introduce an unnecessary new notation (later becoming widely known as the Hoare triple), and to present the logic of programming as a set of proof rules rather than as algebraic axioms. What a shame!

In [8] I followed my original inspiration to present programming as a form of algebra. My objective in writing the paper was to unify various theories that were then current, for example, those due to [5,22,2]. But I did not think of including in the unification my own theory of Hoare logic, or the algebraic laws used for concurrent programming, for example in [1,13] or the operational semantics based on [11].

This essay has been written to rectify these three omissions. It is offered not as an apology for my past mistakes, because in science mistakes are always a necessary prelude to success. It is offered rather as a key to the correction of the mistakes, and perhaps also as a pointer to a broad and attractive direction of future research, leading eventually to beneficial application in professional programming practice.

References

1. Roscoe, A.W.: Laws of occam programming, Tech Mon PRG-53. Oxford University (1986)
2. Tarski, A.: On the Calculus of Relations. J. Symbolic Logic 6(3), 73–89 (1941)
3. Hoare, T.: Unifying Semantics for Concurrent Programming (to appear, 2013)
4. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A Theory of Communicating Sequential Processes. JACM 31, 560–599 (1984)
5. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Englewood Cliffs (1976)
6. Pratt, V.: The Pomset Model of Parallel Processses: Unifying the Temporal and the Spatial. Stanford University, STAN-CS-85-1049 (January 1985)
7. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Comm. ACM 12(10), 576–580 (1969)

8. Hoare, C.A.R., Hayes, I.J., He, J., Morgan, C., Roscoe, A.W., Sanders, J.W., Sørensen, I.H., Spivey, J.M., Sufrin, B.: Laws of Programming. Comm. ACM 30(8), 672–687 (1987)
9. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall (1998)
10. O'Hearn, P.W.: Resources, Concurrency and Local Reasoning. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 49–67. Springer, Heidelberg (2004)
11. Plotkin, G.D.: A Structural Approach to Operational Semantics. DAIMI FN-16 Computer Science Department Aarhus University (1981)
12. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
13. Hennessy, M.: Algebraic Theory of Processes. MIT Press (1988)
14. Baeten, J.C.M., Basten, T., Reniers, M.A.: Process Algebra: Equational Theories of Communicating Processes. Cambridge University Press (2010)
15. Wehrman, I., Hoare, C.A.R., O'Hearn, P.: Graphical models of separation logic. IPL 109(17), 1001–1004 (2009)
16. Hoare, T., Wickerson, J.: Unifying Models of Data Flow. Software and Systems Safety, pp. 211–230. IOS Press (2011)
17. Hoare, C.A.R., Hussain, A., Möller, B., O'Hearn, P.W., Petersen, R.L., Struth, G.: On Locality and the Exchange Law for Concurrent Processes. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 250–264. Springer, Heidelberg (2011)
18. Hoare, T., Moeller, B., Struth, G., Wehrman, I.: Concurrent Kleene Algebra and its Foundations. J. Log. Algebr. Program. 80(6), 266–296 (2011)
19. Hoare, T., van Staden, S.: The Laws of Programming Unify Process Calculi. In: Gibbons, J., Nogueira, P. (eds.) MPC 2012. LNCS, vol. 7342, pp. 7–22. Springer, Heidelberg (2012)
20. van Staden, S., Hoare, T.: Algebra Unifies Operational Calculi. In: Wolff, B., Gaudel, M.-C., Feliachi, A. (eds.) UTP 2012. LNCS, vol. 7681, pp. 88–104. Springer, Heidelberg (2013)
21. Back, R.-J., von Wright, J.: Refinement Calculus. Springer Graduate Texts in Computer Science (1998)
22. Morgan, C.: Programming from Specifications. Prentice Hall International (1990)

Ours *Is* to Reason Why

Cliff B. Jones, Leo Freitas, and Andrius Velykis

School of Computing Science, Newcastle University
{cliff.jones,leo.freitas,andrius.velykis}@ncl.ac.uk

Abstract. It is now widely understood how to write formal specifications so as to be able to justify designs (and thus implementations) against such specifications. In many formal approaches, a "posit and prove" approach allows a designer to record an engineering design decision from which a collection of "proof obligations" are generated; their discharge justifies the design step. Modern theorem proving tools greatly simplify the discharge of such proof obligations. In typical industrial applications, however, there remain sufficiently many proof obligations that require manual intervention that an engineer finds them a hurdle to the deployment of formal proofs. This problem is exacerbated by the need to repeat proofs when changes are made to specifications or designs. This paper outlines how a key additional resource can be brought to bear on the discharge of proof obligations: the central idea is to "learn" new ways of discharging families of proof obligations by tracking one interactive proof performed by an expert. Since what blocks any fixed set of heuristics from automatically discharging proof obligations is issues around data structures and/or functions, it is expected that what the system can learn from one interactive proof will facilitate the discharge of significant "families" of recalcitrant proof tasks.

1 The Challenge

The stimulus for this research has been observing engineers in industry using "formal methods". For example, in the DEPLOY project [24,19], engineers from six companies attempted to use theorem proving tools. It became clear that the number of proof obligations that were not discharged automatically presented a disincentive for deploying formal specifications and design verification.

In a UK project known as AI4FM [23,4,6,5,13] we have set ourselves the challenge of improving the effectiveness of formal methods by using Artificial Intelligence (AI) approaches to remove some of the bottlenecks in the construction of formal proofs.[1] More specifically, we see scope for learning proof strategies from experts.

[1] The title of this paper plays on the jingoistic poem *The Charge of the Light Brigade* by Tennyson which includes:

> Theirs not to reason why,
> Theirs but to do and die

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 227–243, 2013.

The target of the project is the sort of proofs needed in typical justifications of the design and implementation of software systems; it does not aspire to learn how mathematicians prove deep theorems. This section explains the desirability of meeting this challenge and our approach thereto; the body of the paper (Sections 2 and 3) presents an architecture for a system under construction and the rationale for that model. Section 4 discusses the status of this ongoing research.

1.1 Setting Out the Challenge

Increasingly, organisations are recognising that formal descriptions of systems are a useful intermediate step between informal requirements and detailed design.[2] A crucial advantage of a system description in a tractable formal notation is that it provides a basis for the construction of a correctness argument for design. A stepwise process can therefore provide a chain of argument which shows that an implementation (under assumptions about adherence to the semantics of the implementation language) satisfies the initial specification.

"Posit and prove" development methods such as VDM [9] or Event-B [1] allow engineers to make intuitive design steps from which are generated "proof obligations" (POs) whose discharge justifies the posited design choices. Tailored automatic theorem proving tools such as those available in the "Rodin Tools" [18], or general purpose theorem provers (TPs) such as Isabelle [16,17], can automatically discharge a high percentage of proof obligations for industrial scale problems; but a small percentage of a large number still leaves an unwelcome interactive theorem proving load for industrial engineers who are neither specifically trained as logicians nor do they obtain the same enjoyment that an academic might from polishing off proofs. In an example from the EU-funded DEPLOY project [24], around 500 POs were generated just to show that a model was consistent; 80% of these were discharged automatically by the Rodin Tools but this left the engineers facing about a hundred interactive proofs. There is little point in arguing whether some other TP tool would discharge a larger proportion of such POs — with a fixed set of heuristics, some POs will always remain undischarged. This issue is delaying –and will continue to limit– wider use of "formal methods".

Further investigation of the hundred remaining POs does, however, offer some more encouraging news: on examination, there were no more than five ideas which made it easy to discharge all of the residual POs. It is this observation –which is echoed in many other examples– that leads us to the approach being followed in the AI4FM project. It is clear that major research progress has been made in discovering general purpose TP heuristics; on the other hand, undecidability results limit hubris and experience suggests that it is properties that are specific to the data structures and functions of an application that make

[2] This paper does not address the transition from requirements to formal specifications: the issues around understanding requirements are addressed in Jackson's "Problem Frame Approach" [8]; ways of determining specifications of control systems from requirements on overall system behaviour are considered in [14].

proofs go through. The resource that our project hopes to tap is the ability of an expert to spot these specific properties. Having once identified them, the system should then absorb them and use them to discharge further similar proofs. This has two major payoffs: not only is the expert's time not wasted performing encores with proofs that are somehow "in the same family"; but there is also a higher probability that proofs will be found automatically after (inevitable) minor changes to specifications or designs.

The hypothesis of the AI4FM project is:

> *Enough information can be automatically extracted from an interactive proof that examples of the same class can be proved automatically*

As discussed below, this information might either be high level strategies or be captured as lemmas.

Before moving on to our approach and a model of the proposed system, there are a few issues worth putting to rest lest they are of concern to readers. First and foremost, we are fully aware of the considerable power of modern TP systems and their tactic languages: as indicated below and in a companion technical report [22],[3] the initial action with any PO is to pass it to at least one TP system. A particularly encouraging experiment is described in Matthias Schmalz's ETH PhD thesis [20] where he shows that a tailored version of Isabelle manages to discharge a higher percentage of the POs than the built-in TP tools of the Rodin Toolset. (The case study is of significant size and is taken from a different DEPLOY partner than the example discussed above.)

It is also worth noting some factors that must qualify reported figures about "automatically" discharging POs. One such factor is alluded to in reporting Schmalz's experiment: he was very careful to split the collection of POs into training and evaluation sets but the fact remains that Isabelle was hand tailored to the training set. A useful view of the AI4FM hypothesis is that we are aiming to automate that tailoring based on monitoring the activities of an expert in discharging a small number of intransigent POs.

Another significant caveat to any claimed figures on "percentages of automatically discharged POs" concerns reformulations of the models. To take one source, in [1] there are some beautifully staged developments that are split into many steps with the effect that the POs are relatively easy to discharge. Even were it the case that the published developments actually represent the author's first attempts, it must be remembered that the author is both an expert and understood thoroughly the strengths/weaknesses of the TPs in the Rodin Tools. An engineer hoping to deploy the same tools is neither likely to be so expert nor wish to reformulate rather larger (than in *any* textbook) models to make the task of the TP system easier. Abrial's book is chosen for comparison because his proofs have been, laudably, discharged using tools. One of the current authors also extols the advantages of stepwise development (see for example, [9,10,12]) so the

[3] Although frequent references are made to this technical report, the current paper should be self-contained. The longer report contains details of an example that is large enough that it cannot be covered in a paper of this length.

question of how much reformulation is in order is clearly one of degree. Without being able to provide precise metrics, the position taken in the AI4FM project is that "posit and prove" developments should split a design so that each step reflects a clear design decision. If –as often happens– that leaves undischarged POs, the problem should be tackled by introducing concepts during theorem proving rather than by interposing extra steps of development. (This issue is discussed further –and illustrated– in the companion technical report [22].)

One final comment is in order: the current authors are (painfully) aware that many POs actually represent unprovable conjectures. The role of model-checking approaches such as "ProB" [26] in detecting mistakes is invaluable.

1.2 Tackling the Challenge

The objectives of the AI4FM project were recognised by its proposers and reviewers alike as being "ambitious". Of course, the objectives might not just be ambitious — they might be unachievable. At a minimum, the project has to design an unusual way of describing high-level strategies. Here, our experience suggests that the design of such a "language" is more likely to succeed if it is driven from the "state" of the language (rather than its syntax).[4]

A relevant experience for the current project is that which created *mural* [11]. A prime objective of the earlier project was to devise a style of interacting with a TP system that kept the user fully aware of the status of a proof and able to make any sort of forward, backward or intermediate ("cut") step that he or she wished. For its time, this was also considered to be ambitious. In the project that built the *mural* system, we spent a long time iterating versions of its formal description; in fact, project members role-played many versions before any thought was given to actual implementation. The ratio of design time to (initial) implementation was more than four to one.[5] The Newcastle AI4FM team is taking a similar approach. What follows is the n'th iteration of a model of the architecture of a system that we are only now beginning to implement. The following sections (2 and 3) represent an attempt to provide a readable introduction to the model that is summarised in Appendix A — Section 4 includes a discussion of some alternatives.

2 Organising Theories

The project will only succeed if a way is found of expressing high-level strategies; moreover, such strategies need to be generalised from instances of lower-level steps. We expect to use a strongly declarative "language" rather than strings of instructions to a TP system. It is argued in Section 3.1 that this will only be possible if a "top-down" hierarchical view of proofs is taken. We anticipate that

[4] Christopher Strachey argued for working out what you want to say before worrying about how to say it.

[5] An additional bonus was that the model was kept up-to-date during the evolution of the system — it is published as [11, Appendix C].

the system will be able to track the overall process by which a user constructs an interactive proof and that "parsing" the detail against the user's "intent" will be possible.

Thus we intend that our proposed system notes the intent of an expert user so as to match this against other tasks. These ideas are described in Section 3. The current section builds up an understanding of the architecture of the AI4FM system in which information about proofs themselves is stored.

The ideas have been tested on a number of examples and the companion technical report [22] contains a non-trivial development (the discussion there and the fact that we made mistakes that were uncovered whilst discharging the POs support the view that, although the example is smaller than the industrial examples that are our final target, it is significantly more challenging than any that would fit in a paper of this length).

2.1 Bodies of Knowledge

The accumulated knowledge in AI4FM is stored in a collection of named bodies (in the sense of "body of knowledge").[6]

$$\Sigma :: bdm : BdId \xrightarrow{m} Body$$
$$\ldots$$

These bodies can be related to each other in various ways — this topic is discussed in Section 3.2. There will be bodies of knowledge about general mathematical theories such as set theory (cf. Section 2.2); there will also be bodies that relate to a specific application (cf. Section 2.3). Thus far, this is a conventional structure but it is one into which more novel concepts are embedded.

2.2 Base Theories (as *Body* Objects)

Consider, say, the *Body* for sequences of "locations"[7] as in the model in [22, Appendix B]. The *BdId* will be some memorable name such as LocSeq. It will "use" both the theory for *Loc* and that for $\mathbb{N}$ (for indexing and the result of **len** s). Within the theory, there will be a series of functions such as $s(i)$, $s_1 \curvearrowright s_2$, **hd** s, **tl** s (operators are viewed as functions written in an infix –or even mixfix– notation). A *FnDefn* will contain the signature of the function and, optionally, an explicit definition in terms of more basic operators. So, "append" might be an operator characterised by axioms; whereas *rev* might be defined by a recursive definition. Thus far:

$$\begin{aligned} Body :: uses &: BdId\text{-}\mathbf{set} \\ &\ldots \\ functions &: FnId \xrightarrow{m} FnDefn \\ &\ldots \end{aligned}$$

[6] Records, mappings, sets etc. are defined in VDM notation — this should present no real hurdle but readers who want to check details are referred to [10].

[7] Obviously, we intend to handle polymorphism — but this is not covered in the current paper. The approach will almost certainly follow that worked out in [11].

$$\begin{array}{lll} \mathit{FnDefn} :: & \mathit{type} & : \mathit{Signature} \\ & \mathit{defn} & : [\mathit{Definition}] \end{array}$$

2.3 Specifications Give Rise to Bodies

As well as general theories, we also expect each user specification to be linked to a *Body* corresponding to its "state". Something like the Overture tool [25] will generate a *Body* for each specification (cf. Appendices of [22] that include a top-level specification and two refinement steps). It is often useful to know the problem domain to which a specification relates — for example, in the RAIL domain, frequent use is made of relations to record track layouts.

$$\begin{array}{lll} \mathit{Body} :: & \cdots & \\ & \mathit{domain} & : \{\text{RAIL}, \text{AUTO}, \ldots\} \\ & \cdots & \end{array}$$

In most industrial cases, the state will be defined as a record. In examples such as those from the industrial partners in the DEPLOY project, states of 20 fields were not unusual — and these states also tended to have lengthy invariants. It might be worth generating theories for any separable sub-states in the sense that data type invariants and/or operations force some fields to be grouped together — other than these constraints, models should be split as far as is possible — each distinct record type will be translated into a body.

Within a body for a specification, a proof obligation generator (POG) will place a *Conjecture* for each PO about the consistency (e.g. invariant preservation) of that single specification. Proof obligations will also be generated corresponding to the claim that one model reifies another (obviously this has to be triggered by the claimed reification link).

2.4 Conjectures

The information in a *Body* that is of use in proofs is the collection of formal results that are built up over the lifetime of that body.

$$\begin{array}{lll} \mathit{Body} :: & \cdots & \\ & \mathit{theory} & : \mathit{ConjId} \xrightarrow{m} \mathit{Conjecture} \\ & \cdots & \end{array}$$

When first generated, a *Conjecture* is actually a proof task. Each such conjecture has hypotheses and a goal both containing judgements. A *Judgement* can be a sequent or an (in-)equation. In addition there can be any number of (attempts at) justifications. Thus:

$$\begin{array}{lll} \mathit{Conjecture} :: & \cdots & \\ & \mathit{hyps} & : \mathit{Judgement}^* \\ & \mathit{goal} & : \mathit{Judgement} \\ & \mathit{justifs} & : \mathit{JusId} \xrightarrow{m} \mathit{Justification} \\ & \cdots & \end{array}$$

$$\mathit{Judgement} = \mathit{Sequent} \mid \mathit{Equation} \mid \mathit{Ordering} \mid \cdots$$

So, for example, $s \in \mathit{LocSeq} \vdash \mathit{rev}(\mathit{rev}(s)) = s$ is likely to be a judgement accompanied by a proof; other judgements will be axioms.

2.5 Justifications

Turning to *Justification*, notice that it is explicitly envisaged that there can be multiple attempts to justify a proof task. When a conjecture is first generated, it will have no justifications. A user might start one proof *Attempt*, leave it aside and try another, then come back and complete the first proof.

Many conjectures will not contain proofs as such. There might for example be an axiom that $\mathbf{hd}\,([a] \frown s) = a$ and a proof of $\mathit{rev}(\mathit{rev}(s)) = s$. Unsurprisingly, a flag AXIOM will be used to mark axioms. Another way in which a justification need not be (a graph of) a logical proof is that it might be copied from some separate TRUSTED source.

In practice, TP tools such as Isabelle and Z/EVES are powerful enough that a user will hardly ever interact at the level of the (natural deduction) laws of the logic itself. So, in fact, the most prevalent examples of *Justification* ought come from the underlying theorem prover. Automatic use of a TP system will be recorded as an instance of *Tool* (and might include the configuration used). Other obvious examples of *Tool* might record the use of a SAT/SMT tool (which could also be used to look for counter examples if the first attempt at proof fails).

$$\mathit{Justification} = \text{AXIOM} \mid \text{TRUSTED} \mid \mathit{Tool} \mid \mathit{Attempt}$$

The idea of *Attempt* is to be able to accommodate (manual) proof steps.

$$\begin{array}{lllll} \mathit{Attempt} & :: & \mathit{rule} & : & \mathit{ConjId} \\ & & \mathit{hyps} & : & \mathit{ConjId}^* \\ & & \mathit{subst} & : & \mathit{Term} \xrightarrow{m} \mathit{Term} \\ & & \mathit{sub\text{-}probs} & : & \mathit{ConjId}\text{-}\mathbf{set} \end{array}$$

Notice an attempt corresponds to one step in a proof: collecting a whole proof requires tracing the attempts at the sub-conjectures. Thus the notion of whether a proof is complete (in the sense of (transitively) relying only on axioms) is a complex recursive predicate. A low-level instance of *Attempt* might record that the (*rule*) on which it is based is "or elimination". More interesting would be the use lemmas.

3 Strategies

As indicated, the aim of the AI4FM project is to support users with the discharge of industrial scale POs. The way in which we expect to extract strategic insight from proofs –possibly undertaken by experts– is described in Section 3.1; selection and replay (with modifications) is covered in Section 3.2. First, the data structures are described.

Strategies reside in the relevant *Body*:

$$\begin{array}{l} \mathit{Body} :: \cdots \\ \qquad \mathit{strats} : \mathit{StrId} \xrightarrow{m} \mathit{Strategy} \end{array}$$

A low level strategy might split a problem into sub-cases; another could reduce an expression to a normal form; an important collection of strategies will be for induction; an interesting strategy might shift the representation of an object of interest to a different body of knowledge.

We have for a long time within the AI4FM project referred to the "why" of strategies and conjectures (and this is the reason for the use of this word in the title of the paper). The point is that it is easier to achieve a high level of strategy re-use if the intent is captured rather than if only a transcript is recorded. The initial conjectures come from POG and their *source* will contain the name of the kind of proof obligation. Conjectures that are generated as sub-problems by a strategy will be marked with the *Why* value of the strategy.

$$\begin{array}{l} \mathit{Conjecture} :: \mathit{source} : \mathit{Origin} \mid \mathit{Why} \\ \qquad\qquad\quad \cdots \end{array}$$

Examples of values for *Why* are given in Section 3.1 (and more are listed in [22]). The set will never be closed so that a user can always add a new concept.

Strategic information needs to represent both "and" and "or" situations. The "or" function is represented by having alternative strategies. For example, we do not explicitly say that three strategies whose *StrId*s are StructuralIndn, $\mathbb{N}$PeanoIndn and $\mathbb{N}$CompleteIndn are options — it's just that their *intent* fields are all likely to be marked as something like HandleUniversal. The selection between alternatives is, in a sense, underneath the covers for the user (it might give rise to limited parallelism).

An "and" split in a *Strategy* records that, in order to justify a conjecture, certain other conjectures must be discharged (although in some cases it will just be a reformulation and generate only one sub-task — e.g. contrapositives of implications, use of an isomorphic model — at the leaves of a strategy there are no sub-tasks). Just as in LCF-like systems, the flip side of *split* is the *justif* which proves that the sub-goals (when discharged) justify the original goal.

$$\begin{array}{lll} \mathit{Strategy} :: \mathit{intent} & : & [\mathit{Why}] \\ \qquad\qquad\quad \mathit{split} & : & \mathit{Conjecture} \to \mathit{Conjecture}\text{-}\mathbf{set} \\ \qquad\qquad\quad \mathit{justif} & : & \mathit{JusId} \\ \qquad\qquad\quad \cdots & & \end{array}$$

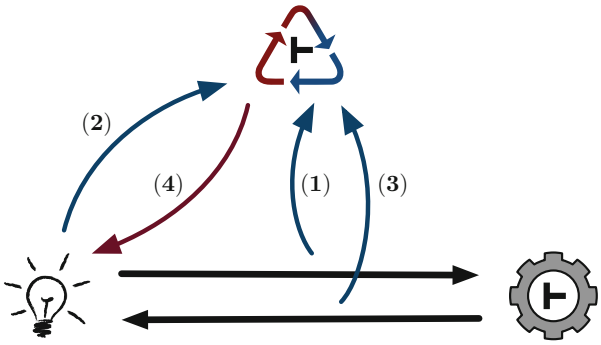
Notice that *split* is a general function, not a mapping. One possibility for the *split* field is that it could contain a text in a language that is interpreted. In contrast, it fits our top-down view better to have high-level derived rules — see the discussion of lemmas in the next sub-section.

After a split is made, the theorem prover of choice will be triggered on each of the generated sub-goals. If they are all discharged, this reinforces the strategy used; if the user hits a dead-end, the likelihood of trying that strategy in similar circumstances is reduced. Furthermore, the expert has to backtrack to some other decision in the proof process.

3.1 Extracting Strategies

As indicated in Section 1.1, over and above general purpose heuristics, the key resource that AI4FM hopes to exploit is to garner information from interactive proofs. It is convenient to talk about this in terms of the interactive proof being undertaken by an expert but it might also be the case that an engineer who is working on PO discharge behaves as the expert — perhaps after some reflection.

The following diagram shows how AI4FM is intended to snoop on the interaction of the expert with the theorem proving system. The symbols stand for the expert (depicted by a lightbulb for inspiration), the TP system (a cogwheel around the turnstile symbol) and at the top of the diagram AI4FM (marked by our trademark for recycling deductions). The basic two way interaction between the expert and the TP system is marked by the horizontal arrows.



The numbered arcs showing interactions with AI4FM are explained as follows:

1. Having a record of why a conjecture is being tackled, the system can attempt to "parse" any interactions initiated by the expert against existing strategies.
2. The expert will be asked to name any new strategies and be invited to mark identifying features.
3. The system can note undischarged goals, record success/failure of strategies; and record the lemmas that are used.
4. The system can suggest strategies to the expert.

This illustrates the primary way in which AI4FM will accrete information. Notice that the aim is to mine the proof *process* which we feel has far more information than just finished –and possibly polished– proofs.

One point that we feel is crucial is the importance of starting the analysis of what the user (expert) is doing from the initial goal (the "top" of the proof): knowing why a conjecture arose is the key to getting an appropriate "parse" of the steps made; looking at the steps alone is a much harder way of determining an expert's intent.[8]

[8] Hearing a seminar on programs that "understand" music prompted the analogy of trying to guess the form of a piece of music a bar at a time versus trying to "parse" it against some expected structure(s).

A failure to discharge a PO will be apparent when one or more conjectures cannot be proved either by the underlying theorem proving systems or any of the available strategies. When such an impasse is reached, an expert might introduce a lemma. Alternatively, the expert might respond by making a different choice in some step of the proof. For now, we assume that this step is captured without immediate generalisation (cf. Section 3.2). The expert is prompted to provide a name (*Why*) for the new idea. In some cases, it will be possible for AI4FM to track that a new strategy specialises an existing one but in the worst case it is certainly worth having the option to add this relationship by hand.

It is also useful to organise a "taxonomy" of strategies. The idea is perhaps best illustrated by an example:

ℕPeanoIndn specialises ℕIndn
ℕCompleteIndn specialises ℕIndn
ℕIndn specialises Indn
StructuralIndn specialises Indn

So the final field of *Strategy* is:

Strategy :: ⋯
specialises : [*StrId*]

There are also what might be thought of as "meta-strategies". One of these is referred to by the second author as "zooming". Given, for example (in technical report [22, B.1]), a proof obligation that involves expressions such as $pre\text{-}NEW0(s, \sigma)$ there are three levels at which the PO can be passed to a theorem prover: as is (with nothing expanded); expansion of the specific predicate $pre\text{-}NEW0$ (about which there are likely to be no lemmas) to terms/operators of set theory; or even an expansion of everything down to predicate calculus. In general, proofs are clearest to a user if they can be conducted with least expansion. In fact, a genuine expert will often "anti-zoom" and prove results at a more general level than their original expression.

A frequent contribution from an expert is to spot that a lemma will provide the clue that makes automatic theorem proving succeed. This approach is seen most clearly in the "waterfall" of ACL2 [15] but it also applies to LCF-style theorem provers such as Isabelle. Lemmas essentially bundle up steps in an argument so that one application of a strategy moves a proof many steps towards its goal. In this section, we assume that lemmas are captured in exactly the form in which they are used; Section 3.2 indicates one source of generalisation. We are also investigating other ways of spotting generalisations at the point of lemma capture. A way of relating bodies of knowledge is described in the next section and this is one technique by which patterns between lemmas can be utilised.

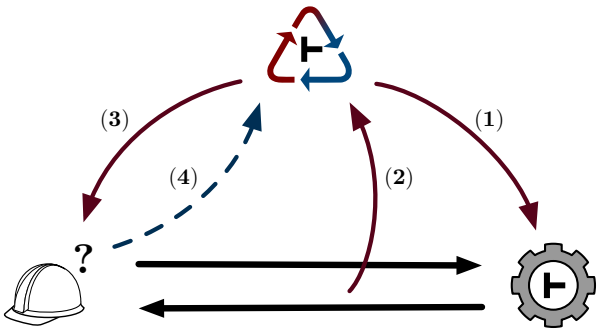
3.2 Replaying Strategies

For some specific *Conjecture*, a user might wish to provide a justification. There is, in fact, rather more behind this comment than someone schooled in say Isabelle might expect. First of all, AI4FM will provide many ways of viewing the outstanding proof tasks so that, for example, a user can see all of the unjustified

leaves of the tree from some specific PO. It is also envisaged that a user can opt to provide new proofs for already justified results — this is why *Conjecture* contains a map (*justifs*) to any number of justifications.

The description that follows, however, assumes a single point of focus: a *Conjecture*. If the theorem prover of choice can discharge the conjecture automatically, the justification is recorded as a *Tool* transcript. The more interesting case for AI4FM is where automatic proof fails.

The following diagram shows how AI4FM can assist the interaction of an engineer (marked by a hardhat) with a TP system.



Here again, the extra indexed arcs are explained:

1. The system can replay (possibly modified versions of) strategies that fit the context and have been previously generated in expert mode. As explained below, an attempt is made to order the use of options based on previous success/failure.
2. Success/failure of strategies is noted both to trigger a move to the next option and to adjust weights that will affect future choices. If necessary, failure of the final option will cause the system to backtrack to an earlier point in the proof tree.
3. The system must keep the user informed (especially about backtracks); it might also ask about lemmas.
4. The engineer might be able to assist if automatic attempts (just) fail; alternatively, there might be a need to bring an expert on line.

Given a collection of strategies, we need a way of selecting the one that is most likely to succeed. There are two sources of information. The provenance information in the *source* field of *Conjecture* is discussed above; in addition to the fields listed at the beginning of Section 2.4, a *Conjecture* will contain information about its features:

$$\begin{aligned} \mathit{Conjecture} &:: \cdots \\ &\quad \mathit{match} : \mathit{Features} \end{aligned}$$

A putative shape of *Features* is given in Appendix A — the final list will be chosen after experimentation. The order in which strategies are tried is governed by its *Score* and this is an area where we hope to use some form of "machine

learning". It is therefore crucial that the success or failure of strategies is recorded to adjust the weights in the *rank* function

$$\begin{aligned} &\mathit{Strategy} :: \cdots \\ &\qquad \mathit{rank} : \mathit{Conjecture} \to \mathit{Score} \\ &\qquad \cdots \end{aligned}$$

Another way of ordering strategies is based on the assumption that the most specific strategy should be tried first and a *specialises* field is added to *Strategy* to locate the next more general strategy:

$$\begin{aligned} &\mathit{Strategy} :: \cdots \\ &\qquad \mathit{specialises} : [\mathit{StrId}] \end{aligned}$$

In this case, failure prompts trying the next more general strategy.

A stored strategy might well rely on a lemma and the exact form of the lemma used in the situation from which the strategy was extracted might not fit the context where the strategy is being replayed. So far, we have considered two ways of evolving ("educing") lemmas: one involves looking in related bodies of knowledge; the other attempts to infer a modified lemma from contextual information.

With regard to relationships between bodies of knowledge, Σ contains:

$$\begin{aligned} &\Sigma :: \cdots \\ &\qquad \mathit{bdrels} : (\mathit{BdId} \times \mathit{Relationship} \times \mathit{BdId})\text{-}\mathbf{set} \end{aligned}$$

which stores relationships between bodies of knowledge. Like *Why* itself, this will have to be expandable by the user. Some examples include:

$$\begin{aligned} \mathit{Relationship} = \mathit{Specialisation} \mid \mathit{Morphism} \mid \mathit{Isomorphism} \mid \\ \mathit{Inherits} \mid \mathit{Sub} \mid \mathit{Similarity} \mid \mathit{Difference} \mid \cdots \end{aligned}$$

AI4FM might, for example, have some abstract items in *Body* such as Larch's "collector" [7]; sets, sequences and maps would all then be specialisations of collector. Another abstract item might be "inductable" where the more general knowledge about setting up inductive proofs would reside. *Morphism* and *Isomorphism* will be used for precise mathematical relationships — the latter where results can be used in either direction. *Similarity* will be for less precise connections (fuzzy matches).

To begin with a low-level example of how the relationships between bodies can be used, suppose a strategy for rearranging operators was applied on set operators and the associativity of union was used, then application of the same strategy on sequence operators might generate the need for a lemma for the associativity of concatenation. In this case, one would expect that such a lemma would already be in the appropriate *Body*. A more interesting example might be ways of creating witnesses for existential quantifiers.

The approach of evolving a lemma from the originating proof to match a new context looks to be feasible. Comparing the hypothesis and goals of the original proof task with the lemma that was generated in expert mode ought to provide enough information to tailor a new lemma that fits the context in which the containing strategy is replayed.

If all options for a strategy have proved fruitless, AI4FM will be able to locate a higher point in the "proof tree" and try alternatives from there. The balance of exploring breadth versus resorting to backtracking will have to await experiments.

4 Status and Way Forward

As indicated in Section 1.2, we have already spent a lot of time discussing the architecture of our proposed system in terms of a model whose current version is in Appendix A. Some of the issues already resolved are outlined in Section 4.2; Section 4.3 sketches our immediate priorities; the first sub-section outlines our experiments with case studies.

4.1 Outline of Case Studies

As has been made clear above, the model of the architecture described in the preceding sections and summarised in Appendix A has evolved over experiments with case studies. Experience has shown that relatively little can be learnt from small examples and that many issues only become clear when non-trivial case studies are considered. The specification and development in [22] is not as large as those met –for example– in the DEPLOY project (see [19]) and it has been important that such industrial applications are kept in mind. Apart from the experience of working closely with industrial teams, the authors have direct writing experience of specifications and developments for applications like cash cards, file stores and systems that control access to secure sites.

The management of a free storage "heap" is a well understood computing problem and the development in [12, §7] provided a good starting point for a useful case study.[9] The heap example is too large to cover in detail in a paper of this length which is why a companion technical report [22] is being made available along with all of the formal material in machine readable form.[10] Deviations from the original development of [12, §7] are discussed in [22, Appendix B.3.6]. That report also contains additional observations that come from other case studies.

One thing that has come as a surprise is the degree of difficulty in handling partial terms efficiently in Isabelle. Our surprise might puzzle a reader who knows that the first author has long argued for the use of a "Logic of Partial Functions" [2]; this is clearly an area for more investigation and Schmalz's [20] might offer the approach that fits most closely with Isabelle.

4.2 Alternatives Already Considered

There are some ideas that we have considered but have yet to build into the model. Two such issues are mentioned here: "analysing proof failures" and

[9] In fact, most of the chapters in [12] are non-trivial and usable as case study material.
[10] We have actually undertaken the proofs in both Z/EVES and Isabelle/HOL and the differences are discussed in [22].

"recording negative results". Our project colleagues in Edinburgh have pioneered general ideas about analysing proof failures and more specifically about "rippling" [3]. Superficially, it would be easy to include one or more indicators such as StuckInduction among the values of *Why* but further investigation and experimentation is required to check that this provides a convenient bridge to established –and new– ways of analysing failure.

Prompted by an interesting discussion with Aaron Sloman, we experimented with the idea of recording in the model what might be termed "negative results". The point being that knowing –for example– that, while set union and list concatenation both enjoy properties such as associativity, the latter is not commutative (in general) might provide important clues when trying to replay a strategy from one context in a different body. This idea remains under consideration but for now it is assumed that such negative information is stored as a *Difference* relationship in *bdrels*.

Probably the biggest issue in our discussion on the architecture has been the ways in which one can view the design of a "strategy language". To oversimplify, one can contrast "bottom up" approaches that try to extend the vocabulary of existing tactic languages with the "top down" approach followed in this paper. Of course, the ideal is that these approaches converge and it is clear that the *split* field of *Strategy* in Appendix A could contain texts of an extended tactic language. As indicated in Section 3.1, a strong argument for the top-down approach is that making sense of ("parsing") interactions in expert mode is only possible if the system can track the user's overall objective. It must however be conceded that it would be simpler to capture the tactic-level steps that an expert makes than it will be to parse these against high-level goals. The idea of taking the lower-level approach and adding annotations to such scripts is viewed as a fall-back position; our immediate plan is to tackle the high-level objective.

The model in Appendix A does not order sub-goals in the *split* field of *Strategy*. This assumes that only graph shape matters but we accept that there are cases where order might be important. In fact, we have considered the idea of time stamping each *Conjecture*. We have yet to build this into the model (one can always write a function that drops this information where not needed).

Another option in the model is to be able to locate instances of the use of strategies — but, for the time being at least, the pointers are in the other direction.

As was found in the *mural* project, records (in the VDM sense) can be difficult in that there is really a different theory of selectors and constructors for each record shape. Records are so ubiquitous that we have to do something for them and we do not favour expanding out "axioms" for all of the constructors/selectors. One reason for preferring some built-in handling of records is that theorem provers can actually suffer from an excess of lemmas: the irrelevant clutter makes searching inefficient or even useless.

4.3 Next Steps

Our immediate activity is to tension the model in Appendix A against more non-trivial examples. A trade-off then has to be made as to the point in time when

greater productivity can be achieved by building the model into our on-going tool support activities. The balance here is that it takes almost no time to revise the model on paper but somewhat longer to revise the Eclipse-based tools that are the current work of (mainly) the third author. The first version of an Eclipse interface to Isabelle has been released [21] and it provides an integration platform for our tools and experiments which gather information from interactive proofs.

Even once we switch to slightly slower revision iterations involving the tools, it is our intention to follow the good practice in the *mural* project and to keep the formal model up to date.

Acknowledgements. It is a pleasure to acknowledge the fruitful collaboration with our Scottish colleagues in the AI4FM project. The first author is grateful to Aaron Sloman for a useful discussion at the Birmingham BCTCS meeting. We also derived stimulus from discussions at the Schloß Dagstuhl event (12271) on "AI Meets Formal Software Development". Last, but by no means least, the authors are grateful for the EPSRC funding of the AI4FM project.

References

1. Abrial, J.-R.: The Event-B Book. Cambridge University Press, Cambridge (2010)
2. Barringer, H., Cheng, J.H., Jones, C.B.: A logic covering undefinedness in program proofs. Acta Informatica 21, 251–269 (1984)
3. Bundy, A., Basin, D., Hutter, D., Ireland, A.: Rippling: Meta-level Guidance for Mathematical Reasoning. Cambridge Tracts in Theoretical Computer Science, vol. 56. Cambridge University Press (2005)
4. Bundy, A., Grov, G., Jones, C.B.: Learning from experts to aid the automation of proof search. In: O'Reilly, L., Roggenbach, M. (eds.) AVoCS 2009 – PreProceedings of the Ninth International Workshop on Automated Verification of Critical Systems, CSR-2-2009, pp. 229–232. Swansea University (2009)
5. Bundy, A., Grov, G., Jones, C.B.: An outline of a proposed system that learns from experts how to discharge proof obligations automatically. In: Abrial, J.-R., Butler, M., Joshi, R., Troubitsyna, E., Woodcock, J.C.P. (eds.) Dagstuhl 09381: Refinement Based Methods for the Construction of Dependable Systems, pp. 38–42 (2009)
6. Freitas, L., Jones, C.B.: Learning from an expert's proof: AI4FM. In: Ball, T., Zuck, L., Shankar, N. (eds.) UV 2010 (Usable Verification) (November 2010)
7. Guttag, J.V., Horning, J.J., Garl, W.J., Jones, K.D., Modet, A., Wing, J.M.: Larch: languages and tools for formal specification. Springer (1993)
8. Jackson, M.: Problem Frames: Analyzing and structuring software development problems. Addison-Wesley (2000)
9. Jones, C.B.: Software Development: A Rigorous Approach. Prentice Hall International (1980)
10. Jones, C.B.: Systematic Software Development using VDM, 2nd edn. Prentice Hall International (1990)
11. Jones, C.B., Jones, K.D., Lindsay, P.A., Moore, R.: *mural*: A Formal Development Support System. Springer (1991)
12. Jones, C.B., Shaw, R.C.F. (eds.): Case Studies in Systematic Software Development. Prentice Hall International (1990)

13. Jones, C.B., Grov, G., Bundy, A.: Ideas for a high-level proof strategy language. In: Dutertre, B., Saidi, H. (eds.) AFM 2010 (Automated Formal Methods) (July 2010)
14. Jones, C.B., Hayes, I.J., Jackson, M.A.: Deriving specifications for systems that are connected to the physical world. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 364–390. Springer, Heidelberg (2007)
15. Kaufmann, M., Manolios, P., Strother Moore, J.: ACL2 Computer-Aided Reasoning: An Approach. University of Austin Texas (2009)
16. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
17. Paulson, L.C.: Isabelle. LNCS, vol. 828. Springer, Heidelberg (1994)
18. Rodin: The Rodin Tools can be downloaded from SourceForge (2008), `http://www.sourceforge.net/projects/rodin-b-sharp`
19. Romanovsky, A., Thomas, M. (eds.): Industrial Deployment of System Engineering Methods. Springer (2013)
20. Schmalz, M.: Formalising the Logic of Event-B. PhD thesis, ETH, Zuerich (2012)
21. Velykis, A.: Isabelle/Eclipse (2013), `http://andriusvelykis.github.io/isabelle-eclipse`
22. Velykis, A., Freitas, L., Jones, C.B., Whiteside, I.: How to say why (in AI4FM). Technical Report CS-TR-1376, Newcastle University (March 2013)
23. WWW. AI4FM (February 2013), `http://www.ai4fm.org`
24. WWW. Deploy project web site (February 2013), `http://www.ncl.ac.uk/computing/research/project/3625`
25. WWW. Overture (March 2013) `http://www.overturetool.org`
26. WWW. The ProB animator and model checker (February 2013), `http://www.stups.uni-duesseldorf.de/ProB`

A Summary of "Model"

$$\begin{array}{lll}\Sigma :: & \mathit{bdm} & : \mathit{BdId} \xrightarrow{m} \mathit{Body} \\ & \mathit{bdrels} & : (\mathit{BdId} \times \mathit{Relationship} \times \mathit{BdId})\text{-}\mathbf{set}\end{array}$$

$$\begin{array}{lll}\mathit{Body} :: & \mathit{uses} & : \mathit{BdId}\text{-}\mathbf{set} \\ & \mathit{domain} & : \{\textsc{Rail}, \textsc{Auto}, \ldots\} \\ & \mathit{functions} & : \mathit{FnId} \xrightarrow{m} \mathit{FnDefn} \\ & \mathit{theory} & : \mathit{ConjId} \xrightarrow{m} \mathit{Conjecture} \\ & \mathit{strats} & : \mathit{StrId} \xrightarrow{m} \mathit{Strategy}\end{array}$$

$$\begin{array}{lll}\mathit{FnDefn} :: & \mathit{type} & : \mathit{Signature} \\ & \mathit{defn} & : [\mathit{Definition}]\end{array}$$

$$\begin{array}{lll}\mathit{Conjecture} :: & \mathit{source} & : \mathit{Origin} \mid \mathit{Why} \\ & \mathit{hyps} & : \mathit{Judgement}^* \\ & \mathit{goal} & : \mathit{Judgement} \\ & \mathit{justifs} & : \mathit{JusId} \xrightarrow{m} \mathit{Justification} \\ & \mathit{match} & : \mathit{Features}\end{array}$$

$$\mathit{Judgement} = \mathit{Sequent} \mid \mathit{Equation} \mid \mathit{Ordering} \mid \cdots$$

$$\mathit{Justification} = \textsc{Axiom} \mid \textsc{Trusted} \mid \mathit{Attempt} \mid \mathit{Tool}$$

$$\begin{array}{lll}\mathit{Attempt} :: & \mathit{rule} & : \mathit{ConjId} \\ & \mathit{hyps} & : \mathit{ConjId}^* \\ & \mathit{subst} & : \mathit{Term} \xrightarrow{m} \mathit{Term} \\ & \mathit{sub\text{-}probs} & : \mathit{ConjId}\text{-}\mathbf{set}\end{array}$$

$$\mathit{Tool} = \cdots$$

Could include info about *blocks*: *ConjId*-**set**

$$\begin{array}{lll}\mathit{Strategy} :: & \mathit{intent} & : [\mathit{Why}] \\ & \mathit{split} & : \mathit{Conjecture} \to \mathit{Conjecture}\text{-}\mathbf{set} \\ & \mathit{justif} & : \mathit{ConjId} \\ & \mathit{rank} & : \mathit{Conjecture} \to \mathit{Score} \\ & \mathit{specialises} & : [\mathit{StrId}]\end{array}$$

$$\begin{array}{lll}\mathit{Features} :: & \mathit{mainTps} & : \mathit{BdId}\text{-}\mathbf{set} \\ & \mathit{mainFns} & : \mathit{FnId}\text{-}\mathbf{set} \\ & \mathit{other} & : \cdots\end{array}$$

$$\mathit{Origin} = \mathit{Token}$$

$$\mathit{Why} = \mathit{Token}$$

$$\mathit{Relationship} = \mathit{Specialisation} \mid \mathit{Morphism} \mid \mathit{Isomorphism} \mid \mathit{Inherits} \mid \mathit{Sub} \mid \mathit{Similarity} \mid \mathit{Difference} \mid \cdots$$

Optimal Bounds for Multiweighted and Parametrised Energy Games

Line Juhl[1], Kim Guldstrand Larsen[1], and Jean-François Raskin[2]

[1] Aalborg University, Department of Computer Science, Denmark
{linej,kgl}@cs.aau.dk
[2] Université Libre de Bruxelles, Belgium
jraskin@ulb.ac.be

Abstract. Multiweighted energy games are two-player multiweighted games that concern the existence of infinite runs subject to a vector of lower and upper bounds on the accumulated weights along the run. We assume an unknown upper bound and calculate the set of vectors of upper bounds that allow an infinite run to exist. For both a strict and a weak upper bound we show how to construct this set by employing results from previous works, including an algorithm given by Valk and Jantzen for finding the set of minimal elements of an upward closed set. Additionally, we consider energy games where the weight of some transitions is unknown, and show how to find the set of suitable weights using the same algorithm.

1 Introduction

Energy games have recently attracted considerable attention [1–9]. An energy game is played by two players on a weighted game automaton. Player 1 wins if she has a strategy such that all infinite runs respecting this strategy has nonnegative accumulated weight at all times. A variant of energy games furthermore requires an upper bound that the accumulated weight must stay below at all times in order for Player 1 to win. The upper bound can also be weak, implying that all accumulated weights going above are simply truncated. As embedded systems are often resource-constrained systems exhibiting a reactive behaviour, energy games are relevant for ensuring that the resource of the system never becomes unavailable no matter the choices of the environment. Multiweighted energy games, where the weights of the automaton are vectors, are useful for modelling systems that depend on more than one resource.

In this paper we consider multiweighted energy games with unknown upper bound (both strict and weak) and fixed initial value. When considering the existence of a vector of upper bounds such that Player 1 is winning, it is from an engineering viewpoint relevant to construct the actual vector instead of giving a boolean answer to the problem. We therefore seek to construct the exact set of upper bounds that make Player 1 win the energy game. We will denote such upper bounds as winning. For both types of upper bounds it is clear that if some vector of upper bounds is winning, then also coordinate-wise larger vectors are

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 244–255, 2013.

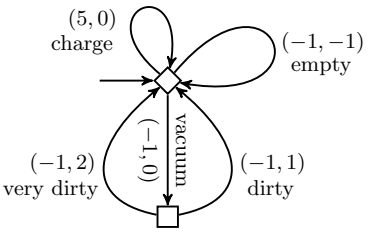
winning. In order to characterise the set of winning upper bounds, it is thus sufficient only to find the smallest vector of winning upper bounds. However, $\leq$ is not a total order on $\mathbb{Z}^k$ for $k > 1$, so instead of a unique smallest vector we search for the set of smallest incomparable vectors winning for Player 1.

To motivate the study, let us consider a small example of an automatic vacuum cleaner. The machine has a rechargeable battery and a container for the dust it collects. As we are interested in a behaviour that never empties the battery nor completely fills the dust container, it can be modelled using a 2-weighted energy game as seen in Fig. 1a. The vector attached to each transition denotes the change in battery (first coordinate) and container level (second coordinate). The diamond state is controlled by Player 1 while the square state is controlled by the environment, as the vacuum cleaner does not control how dirty the floor is when vacuuming.

The lower bound of the two resources are naturally 0, while the upper bound corresponds to the size of the battery and container, respectively. For a manufacturer it is useful to know what size she can possible make the battery and the container in order to ensure an infinite run. The set of minimal winning upper bounds consists in this case of the vectors $(6, 2)$ and $(5, 3)$. The upper bound vector $(6, 2)$ keeps the container as small as possible, while the upper bound vector $(5, 3)$ keeps the battery as small as possible. Surely, the first coordinate of an upper bound cannot be smaller than 5, as charging adds 5 to the accumulated weight in the first coordinate. Similarly, the second coordinate cannot be smaller than 2, as a very dirty floor adds 2 to the accumulated weight in the second coordinate. The winning strategy for Player 1 can be seen in Fig. 1b for $(6, 2)$ and Fig. 1c for $(5, 3)$. Any vector larger than one of the minimal vectors will also serve as a winning upper bound.

Contributions. For multiweighted energy games with an unknown upper bound (both strict and weak) and fixed initial value we calculate the set of minimal upper bounds such that the energy game is winning. For a strict upper bound we make use of results from [3] and [9] in order to construct the set, yielding an algorithm running in $2k$-exponential time. In the case of a weak upper bound we utilise an algorithm given by Valk and Jantzen in [10], that constructs the set of minimal elements of an upward-closed set (the so-called Pareto frontier), by showing that the preconditions for applying the algorithm are fulfilled. The relevant definitions are given in Section 2, while Section 3 and Section 4 treat the cases of a weak and a strict upper bound, respectively.

Furthermore we study a related problem in Section 5, where we consider multiweighted energy games where both the initial value and the upper bound (if any) are known, but where some weights of the transitions are unknown. We call these parametrised transitions. We here seek to characterise the set of possible evaluations for the parameters such that Player 1 can win the energy game. For a weak upper bound, it is again sufficient to construct the set of minimal evaluations such that Player 1 is winning, and we are once again able to apply the algorithm from [10] to construct the set.



(a) A vacuum cleaner 2-weighted game

if *battery* ≥ 1 and *container* ≥ 1
  then empty
else if *battery* ≥ 2 and *container* $= 0$
  then vacuum
else charge

(b) A winning strategy for Player 1 with upper bound $(6, 2)$

if *battery* $= 0$
  then charge
else if *battery* ≥ 2 and *container* ≤ 1
  then vacuum
else empty

(c) A winning strategy for Player 1 with upper bound $(5, 3)$

Fig. 1. A vacuum cleaner example

Related Work. Previously energy games have been considered in different settings. One-weighted energy games with both upper and lower bounds were defined in [2]. Here they study the existence of a winning strategy for Player 1 for a fixed initial value and fixed upper and lower bound and provide bounds on the complexity for the identified problems both in a timed and untimed setting. The paper [9] extends the results from [2] to the multiweighted case. The work of [7] treats multiweighted energy games with only a lower bound and show that deciding whether there exists a vector of initial values for the resources such that Player 1 can win the energy game is coNP-complete and that only finite-memory strategies are sufficient. In [3] they give a procedure running in $(k-1)$-exponential time that calculates the Pareto frontier of winning initial vectors in multiweighted energy games with k weights, a lower bound and unary weights on transitions (vector addition systems with states). For energy games with imperfect information and fixed initial value, the paper [8] proves decidability of the problem, but undecidability in case the initial value is not fixed.

2 Multiweighted Energy Games

In this paper, we let $\mathbb{Z}$ and $\mathbb{N}$ denote the sets of all integers and all nonnegative integers, respectively. We define $\mathbb{N}_\omega$ as $\mathbb{N} \cup \{\omega\}$, where ω is a new element modelling an arbitrary nonnegative integer. Thus $\omega > m$ for any $m \in \mathbb{N}$.

For two k-dimensional vectors $\bar{v}, \bar{v}' \in \mathbb{N}_\omega^k$ we use the notation $\bar{v}[i]$ to denote the ith coordinate of the vector $\bar{v}$ $(1 \leq i \leq k)$ and write $\bar{v} \leq \bar{v}'$ if $\bar{v}[i] \leq \bar{v}'[i]$ for all $i \in \{1, \ldots, k\}$. We define the sum of two vectors as the coordinate-wise sum,

i.e. $\bar{v} + \bar{v}' = (\bar{v}[1] + \bar{v}'[1], \ldots, \bar{v}[k] + \bar{v}'[k])$. The notation $\bar{0} = (0, \ldots, 0)$ is used to denote the vector of all zeros and $\overline{\infty} = (\infty, \ldots, \infty)$ as the ditto for ∞.

A set $K \subseteq \mathbb{N}^k$ is said to be *upward closed* if $\bar{x} \in K$ and $\bar{x} \leq \bar{y}$ implies $\bar{y} \in K$. Furthermore we define $min(K)$ as the set of smallest incomparable vectors of K,

$$min(K) = \{\bar{x} \in K \mid \forall\ \bar{y}(\neq \bar{x}) \in K : \bar{y} \not\leq \bar{x}\}\ .$$

We call such a $min(K)$ the *minimal generating set* of K.

It it well-known that such a set of incomparable vectors of natural numbers will be finite (as stated in Dickson's lemma) and unique.

We now define a game with multiple weights as an automaton with dedicated state sets for each player and a transition function decorated with a vector of integers.

Definition 1. *A k-weighted game is a four-tuple $G = (S_1, S_2, s_0, \longrightarrow)$, where S_1 and S_2 are finite, disjoint sets of existential and universal states, respectively, $s_0 \in S_1 \cup S_2$ is the start state and $\longrightarrow \subseteq (S_1 \cup S_2) \times \mathbb{Z}^k \times (S_1 \cup S_2)$ is a finite multiweighted transition relation.*

We write $s \xrightarrow{\bar{w}} s'$ whenever $(s, \bar{w}, s') \in \longrightarrow$. In the following we consider only non-blocking automata, i.e. for every $s \in S_1 \cup S_2$ we have $s \xrightarrow{\bar{w}} s'$ for some $\bar{w} \in \mathbb{N}^k$ and $s' \in S_1 \cup S_2$.

Definition 2. *A configuration in a k-weighted game $G = (S_1, S_2, s_0, \longrightarrow)$ is a pair $(s, \bar{v})$ such that $s \in S_1 \cup S_2$ and $\bar{v} \in \mathbb{Z}^k$.*

A weighted run π in G restricted to a weak upper bound $\bar{b} \in (\mathbb{N} \cup \{\infty\})^k$ is an infinite sequence of configurations $(s_0, \bar{v}_0), (s_1, \bar{v}_1), \ldots$ such that for all $i \geq 0$ we have $s_i \xrightarrow{\bar{w}_i} s_{i+1}$ and $\bar{v}_{i+1}[j] = \min\{\bar{b}[j], \bar{v}_i[j] + \bar{w}_i[j]\}$ for all $j \in \{1, \ldots, k\}$.

By $\mathsf{WR}_{\bar{b}}(G)$ we denote all weighted runs in G with weak upper bound $\bar{b}$ starting from the initial state. Let π_i denote the ith configuration of a weighted run π.

As we are concerned with games we need a notion of a strategy for a player.

Definition 3. *A strategy for Player $i \in \{1, 2\}$ in a k-weighted game $G = (S_1, S_2, s_0, \longrightarrow)$ restricted to some weak upper bound $\bar{b}$ is a mapping σ assigning a configuration $(s, \bar{v})$ to any finite prefix of a weighted run in $\mathsf{WR}_{\bar{b}}(G)$ of the form $(s_0, \bar{v}_0), \ldots, (s_j, \bar{v}_j)$ where $s_j \in S_i$ such that $(s_0, \bar{v}_0), \ldots, (s_j, \bar{v}_j), (s, \bar{v})$ is a prefix of a weighted run in $\mathsf{WR}_{\bar{b}}(G)$.*

We say that a weighted run $(s_0, \bar{v}_0), (s_1, \bar{v}_1), \ldots$ respects a strategy σ of Player i if $\sigma((s_0, \bar{v}_0), \ldots, (s_j, \bar{v}_j)) = (s_{j+1}, \bar{v}_{j+1})$ for all $s_j \in S_i$.

We can now define the following three notions of winning vectors.

GL: Given a k-weighted game G, a vector $\bar{v}_0 \in \mathbb{N}^k$ *wins* the (multiweighted) energy game with lower bound (GL) if there exists a winning strategy σ for Player 1 such that any weighted run $(s_0, \bar{v}_0), (s_1, \bar{v}_1), \ldots \in \mathsf{WR}_{\overline{\infty}}(G)$ respecting σ satisfies $\bar{0} \leq \bar{v}_i$ for all $i \geq 0$.

GLW: Given a k-weighted game G, a vector $\bar{b} \in \mathbb{N}^k$ *wins* the (multiweighted) energy game with lower and weak upper bound (GLW) if there exists a winning strategy σ for Player 1 such that any weighted run $(s_0, \bar{0}), (s_1, \bar{v}_1), \ldots \in \mathsf{WR}_{\bar{b}}(G)$ respecting σ satisfies $\bar{0} \leq \bar{v}_i$ for all $i \geq 0$.

GLU: Given a k-weighted game G, a vector $\bar{b} \in \mathbb{N}^k$ *wins* the (multiweighted) energy game with lower and upper bound (GLU) if there exists a winning strategy σ for Player 1 such that any weighted run $(s_0, \bar{0}), (s_1, \bar{v}_1), \ldots \in \mathsf{WR}_{\overline{\infty}}(G)$ respecting σ satisfies $\bar{0} \leq \bar{v}_i \leq \bar{b}$ for all $i \geq 0$.

Notice that we may allow an initial weight vector $\bar{v}_0$ different from $\bar{0}$. This is evident by adding a new start state with one transition labelled with $\bar{v}_0$ pointing to the old start state.

Define $I = \{\bar{v}_0 \in \mathbb{N}^k \mid \bar{v}_0 \text{ wins GL}\}$, $W = \{\bar{b} \in \mathbb{N}^k \mid \bar{b} \text{ wins GLW}\}$ and $U = \{\bar{b} \in \mathbb{N}^k \mid \bar{b} \text{ wins GLU}\}$ as the winning vectors for GL, GLW and GLU, respectively. The paper [3] constructs the set $min(I)$ using $(k-1)$-exponential time for k-weighted energy games with unary weights.

This paper aims to construct the minimal generating sets of winning weak and strict upper bounds, $min(W)$ and $min(U)$.

Membership Problem. Another interesting question besides constructing the sets of winning vectors for a given game, is the question of membership; given a k-weighted game G and a vector $\bar{b} \in \mathbb{N}^k$ decide whether $\bar{b} \in W$ (or $\bar{b} \in I$ or $\bar{b} \in U$). The membership problem has been addressed in [9] among others. Table 1 (also found in [9]) gives a full overview of the so far obtained decidability and complexity results for the membership problem. In the table two further classifications of a k-weighted game $G = \{S_1, S_2, s_0, \longrightarrow\}$ are made. We say that a game G is existential if $S_2 = \emptyset$ (Player 1 controls all the states) and that G is universal if $S_1 = \emptyset$ (Player 2 controls all the states). The other subdivision concerns the number of weights, namely whether k is 1, fixed (and $k > 1$), or arbitrary.

Note that the complexity increases as more weights are added, apart from the universal case, where all problems lie in P. This is evident since any membership problem on a universal game with k weights can be solved by solving the same problem for each coordinate independently. Another thing to observe is that deciding membership in I is computationally easier than deciding membership in U in the 1-weighted case, even though membership in I is harder than U for an arbitrary number of weights. This stems from the fact that the configuration space for the problems concerning U (and W) is bounded due to the upper bounds, whereas the same a priori does not hold for the problems concerning I. The computational complexity of the problems concerning membership in W seems to follow the computationally easiest of the two other problems.

3 Weak Upper Bound

In this section we study the problem of finding the set $min(W)$ as defined in Section 2.

Table 1. Complexity bounds for the membership problem

Weights	Type	Existential	Universal	Game
One	$\in I$	$\in$ P [2]	$\in$ P [2]	$\in$ UP $\cap$ coUP [2]
	$\in W$	$\in$ P [2]	$\in$ P [2]	$\in$ NP $\cap$ coNP [2]
	$\in U$	NP-hard [2], $\in$ PSPACE [2]	$\in$ P [2]	EXPTIME-complete [2]
Fixed (k>1)	$\in I$	NP-hard [9], $\in k$-EXPTIME [3]	$\in$ P [9]	EXPTIME-hard [9], $\in k$-EXPTIME [3]
	$\in W$	NP-hard [9], $\in$ PSPACE [9] PSPACE-complete for $k \geq 4$ [9]	$\in$ P [9]	EXPTIME-complete [9]
	$\in U$	PSPACE-complete [9]	$\in$ P [9]	EXPTIME-complete [9]
Arbitrary	$\in I$	EXPSPACE-complete [9]	$\in$ P [9]	EXPSPACE-hard [9], decidable [3]
	$\in W$	PSPACE-complete [9]	$\in$ P [9]	EXPTIME-complete [9]
	$\in U$	PSPACE-complete [9]	$\in$ P [9]	EXPTIME-complete [9]

The paper [10] by Valk and Jantzen contains an algorithm for computing the minimal generating set of an upward closed set $K \subseteq \mathbb{N}^k$ provided that K satisfies a certain decidability criterion.

The decidability question is defined for a set $K \subseteq \mathbb{N}^k$ as a predicate $p_K : \mathbb{N}_\omega^k \to \{\texttt{true}, \texttt{false}\}$ by $p_K(\bar{d}) = (\{\bar{d}' \in \mathbb{N}^k \mid \bar{d}' \leq \bar{d}\} \cap K \neq \emptyset)$. Thus $p_K(\bar{d})$ decides whether or not the set K has any elements in common with the set of vectors smaller than or equal to $\bar{d}$. If $p_K(\bar{d})$ is decidable for any $\bar{d} \in \mathbb{N}_\omega^k$, the algorithm can be applied to compute the minimal generating of K.

We will now argue that the algorithm proposed in [10] is useful for constructing $min(W)$. The set W is upward closed since a weak upper bound $\bar{b} \in \mathbb{N}^k$ that wins GLW ensures that any $\bar{b}' \geq \bar{b}$ will also win GLW. As $min(W)$ is exactly the minimal generating set of W, $min(W)$ can be found using the algorithm in case p_W is decidable.

Lemma 1. *The predicate $p_W(\bar{d})$ is decidable for any $\bar{d} \in \mathbb{N}_\omega^k$.*

Proof. Given a vector $\bar{d} \in \mathbb{N}_\omega^k$ the following procedure will either construct $\bar{b} \in W$ such that $\bar{b} \leq \bar{d}$ or report that no such $\bar{b}$ exists. Let $\bar{d}'$ be the vector $\bar{d}$ where all ω-entries are substituted by ∞.

Starting from the configuration $(s_0, \bar{0})$ we construct a self-covering tree containing prefixes of all weighted runs. Any configuration $(s, \bar{v})$ induces the child $(s', \bar{v}')$ for any $s \xrightarrow{\bar{w}} s'$ such that $\bar{v}'[\ell] = \min\{\bar{d}'[\ell], \bar{w}[\ell] + \bar{v}[\ell]\}$ for all $\ell \in \{1, \ldots, k\}$. The unfolding of the game graph stops for each branch (i.e. weighted run $(s_0, \bar{0}), (s_1, \bar{v}_1), \ldots \in \mathsf{WR}_{\bar{d}'}(G)$) when reaching an i such that either

A. $\bar{v}_i[\ell] < 0$ for some $\ell \in \{1, \ldots, k\}$ or
B. $s_i = s_j$ and $\bar{v}_i \geq \bar{v}_j$ for some $j < i$.

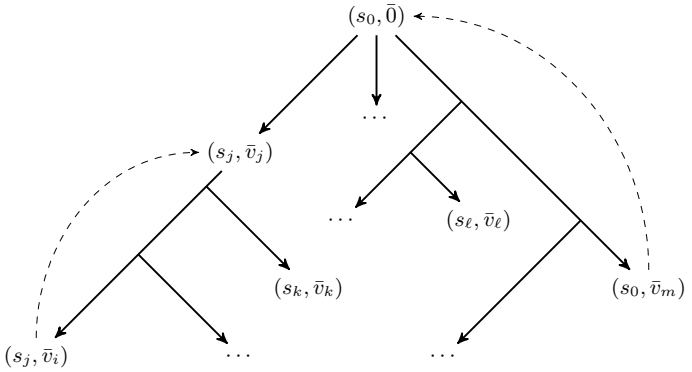


Fig. 2. Self-covering tree

Figure 2 illustrates such a self-covering tree. Here $\bar{v}_i \geq \bar{v}_j$ and $\bar{v}_m \geq \bar{0}$. Any leaf satisfies either A or B.

Notice that since the state set is finite and $(\mathbb{N}^k, \leq)$ is a wqo, such an i exists for all branches and we thus construct a finite tree. In the case of A we mark a leaf configuration $(s_i, \bar{v}_i)$ as losing and in case of B we mark $(s_i, \bar{v}_i)$ as winning. We propagate the marking of the leaves to the configuration $(s_0, \bar{0})$ in the following way, starting with configurations having only leaves as children. If the state of the configuration belongs to Player 1 and at least one child is winning, we mark the configuration as winning. Otherwise it is losing. If the state of the configuration belongs to Player 2 and all children are winning we mark the configuration as winning. Otherwise it is losing. In case $(s_0, \bar{0})$ is losing, $p_W(\bar{d}) = \texttt{false}$, as any weighted run is forced to a losing leaf if Player 2 consistently picks losing children. If $(s_0, \bar{0})$ is winning, we set $p_W(\bar{d}) = \texttt{true}$, as we can construct $\bar{b}$ and a winning strategy σ for Player 1, proving the existence of a winning vector $\bar{b}$ for GLW.

The strategy σ is determined by the tree. For each prefix of each branch $\pi_{\downarrow n} = (s_0, \bar{0}), \ldots, (s_n, \bar{v}_n)$, where $s_n \in S_1$, we let $\sigma(\pi_{\downarrow n}) = (s, \bar{v})$, where $s_n \xrightarrow{\bar{w}} s$ for some $\bar{w}$ such that $\bar{v} = \bar{v}_n + \bar{w}$ and $(s, \bar{v})$ is a winning child of $(s_n, \bar{v}_n)$. For any branch $\pi_{\downarrow m} = (s_0, \bar{0}), \ldots, (s_n, \bar{v}_n), \ldots, (s_m, \bar{v}_m)$, where $s_m = s_n$ and $\bar{v}_m \geq \bar{v}_n$ (a winning leaf) we let $\sigma(\pi_{\downarrow m}) = \sigma(\pi_{\downarrow n})$. Notice that if $(s_n, \bar{v}_n)$ does not have any winning children (or is a losing leaf) a winning strategy will never lead us to this state (and thus any next state can be picked).

It is easy to see that any weighted run respecting σ keeps all accumulated weights nonnegative, since all transitions taken by Player 1 and 2 leads to states marked as winning by the definition of σ. At some point the weighted run will enter a loop that has a nonnegative accumulated weight in all coordinates. Furthermore σ is finitely representable.

The weak upper bound $\bar{b}$ that satisfies $\bar{b} \leq \bar{d}$ and is contained in W can be found by examining the self-covering tree and pruning the tree by removing the branches not respecting σ. For the entries of $\bar{d}$ that are not ω we reuse these entries in $\bar{b}$ and for any remaining ω-entry in dimension ℓ we find the largest

accumulated weight $\max_\ell$ seen in any configuration of the tree in dimension ℓ. Formally

$$\bar{b}[\ell] = \begin{cases} \max_\ell & \text{if } \bar{d}[\ell] = \omega \\ \bar{d}[\ell] & \text{otherwise} \end{cases},$$

for all $\ell \in \{1, \ldots, k\}$. This bound is safe to apply as a weak upper bound, since truncating all weights at this upper bound will not cause the accumulated weights to be negative at any point. □

As Lemma 1 allows us to use the algorithm presented in [10] we get the following corollary.

Corollary 1. *The set $min(W)$ is computable.*

Notice that we can apply the procedure seen in the proof of Lemma 1 for the special case of $\bar{d} = (\omega, \ldots, \omega)$ if we are interested in whether there exists some weak upper bound that wins GLW (e.g. determine whether W is empty or not).

4 Strict Upper Bound

For the case of a strict upper bound we see that U is also an upward closed set, but for deciding $p_U(\bar{d})$ for any $\bar{d} \in \mathbb{N}_\omega^k$ we cannot use the approach presented in the proof of Lemma 1. This is due to the construction of the self-covering tree, where we here cannot stop when reaching a cycle with positive accumulated weight in one of the coordinates, since looping forever (as indicated in Fig. 2 by dashed arrows) will eventually cause one of the strict upper bounds to be violated. For constructing $min(U)$ we instead make use of energy games with no upper bound.

Theorem 1. *The set $min(U)$ is computable in $2k$-exponential time.*

Proof. The paper [9] provides the following useful reduction. Determining whether a given upper bound $\bar{b}$ wins GLU with k weights is polynomial time reducible to determining whether the initial vector $(\bar{0}[1], \ldots, \bar{0}[k], \bar{b}[1], \ldots, \bar{b}[k])$ wins GL with $2k$ weights.

Given a k-weighted game G_k the reduction works by constructing a $2k$-weighted game G_{2k} by doubling the number of weights on each transition of G_k, adding a new start state and letting each new transition have the weight $(\bar{w}[1], \ldots, \bar{w}[k], -\bar{w}[1], \ldots, -\bar{w}[k])$ for any old transition with weight $\bar{w}$. The reduction is seen in Fig. 3, where the circular states denote either a Player 1 or Player 2 state. Now if one of the first k weights goes above $\bar{b}$, one of the last k weights will go below 0.

The paper [3] provides an algorithm running in $(k-1)$-exponential time for constructing the set $min(I)$ for any k-weighted game with only unary updates, that is a game $G = \{S_1, S_2, s_0, \longrightarrow\}$, where each $s \xrightarrow{\bar{w}} s'$ satisfies $\bar{w} \in \{-1, 0, -1\}^k$.

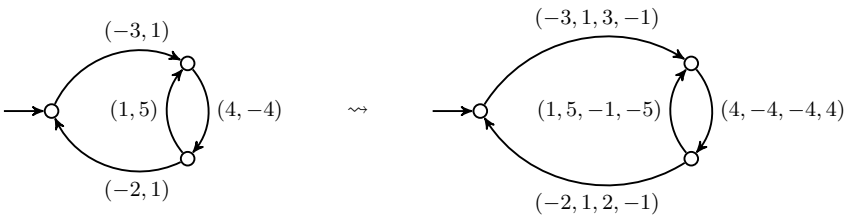


Fig. 3. Example of a reduction from G_k to G_{2k}

We can reduce G_{2k} with arbitrary updates to the unary setting by introducing intermediate transitions that repeatedly add or subtract 1 (causing an exponential blowup in the size of the automaton) and obtain the finite set $min(I)$ by applying the algorithm presented in [3].

Now we can easily construct $min(U)$ for G_k from $min(I)$ for G_{2k} as "subvectors" of the vectors in $min(I)$ with all 0's in the first k coordinates,

$$min(U) = \{\bar{b} \in \mathbb{N}^k \mid (\bar{0}[1], \ldots, \bar{0}[k], \bar{b}[1], \ldots, \bar{b}[k]) \in min(I)\} \ .$$

This procedure presented in [3] runs in $(k-1)$-exponential time for en k-weighted game with unary updates on transitions. Since we in our setting double the number of weights and reduce the arbitrary weights to unary weights, we achieve a procedure running in $2k$-exponential time. □

In case of an energy game with both unknown strict upper bound and unknown initial value, the above proof can as well be applied for finding the set of all pairs of initial values and upper bounds that will win the energy game (this set corresponds to the set $min(I)$ for G_{2k}). The problem of simultaneous synthesis with initial value and strict upper bound can therefore be solved in $2k$-exponential time.

5 Parametrised Transitions

A variant of the problem of parametrised bounds is parametrised transitions. Instead of letting the upper bound or initial value be unknown, we may also consider multiweighted energy games where not all weights of the transitions that gain resources are known. As in the case of an unknown upper bound, we are interested in not only knowing whether there exists an assignment of weights such that Player 1 has a winning strategy in the various energy games, but in constructing the actual set of assignments such that Player has a winning strategy. For no upper bound or a weak upper bound this set is upward closed and can thus be characterised by its minimal generating set. For a strict upper bound this is not the case and we must to represent the set otherwise.

Consider the automatic vacuum cleaner in Fig. 4, where the first coordinate of the weight of the charge transition is unknown (the parameter p). For a strict

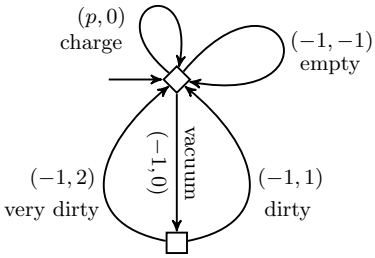


Fig. 4. A parametrised vacuum cleaner example

upper bound of $(5, 3)$ we now seek to compute the set of possible weight assignments to p such that Player 1 has a winning strategy. The smallest possible weight assigned to p is 2 (using the strategy from Fig. 1c). As it turns out, both 3, 4 and 5 as the value of p give rise to a winning strategy for Player 1 (again as seen in Fig. 1c). Surely p cannot be assigned a larger weight than the upper bound in the first coordinate, as this would enable us from charging at any time. The full set of suitable values of p is therefore $\{2, 3, 4, 5\}$.

Formally we let a parametrised k-weighted game $G = \{S_1, S_2, s_0, \longrightarrow\}$ over a set of parameters $P = \{p_1, \ldots, p_\ell\}$ be a game where $\longrightarrow \subseteq (S_1 \cup S_2) \times (\mathbb{Z} \cup P)^k \times (S_1 \cup S_2)$. Given an evaluation function $e : P \to \mathbb{N}$, we let $\longrightarrow_e$ be the set $\longrightarrow$ where any parameter $p_i \in P$ is substituted with its evaluation $e(p_i)$. In case there exists an evaluation function e for a parametrised game G with upper bound $\bar{b}$ such that $\bar{b}$ wins GLU given the game $G_e = (S_1, S_2, s_0, \longrightarrow_e)$, we say that e *wins* GLU with parametrised transitions. The same winning notion can be defined for GLW and GL with parametrised transitions. Given two evaluations e, e', we say that $e \leq e'$ if $e(p_i) \leq e'(p_i)$ for all $i \in \{1, \ldots, \ell\}$. We denote the set of winning evaluations for GLU, GLW and GL with parametrised transitions by U_T, W_T and I_T, respectively. Notice that the sets W_T and I_T are upward closed, implying that these sets can be characterised by their minimal generating set of evaluations, $min(W_T)$ and $min(I_T)$.

For $min(W_T)$ and $min(I_T)$ we as in Section 3 seek to use the algorithm presented in [10] to construct the two sets. The predicates p_{W_T} and p_{I_T} must decide for any parametrised game G whether there exists an evaluation for G in W_T or I_T, respectively.

Lemma 2. *The predicates $p_{W_T}(\bar{d})$ and $p_{I_T}(\bar{d})$ are decidable for any $\bar{d} \in \mathbb{N}_\omega^\ell$.*

Proof. Given a k-weighted game G with parametrised transitions and either a weak upper bound $\bar{b}$ or no upper bound, the existence of a winning evaluation implies the existence of a winning evaluation e that for all $i \in \{1, \ldots, \ell\}$ satisfies $e(p_i) \leq M$, where M is the largest sum obtained by adding all negative weight updates in one coordinate, i.e. $M = \max_{j \in \{1,\ldots,k\}} \left(\sum_{s \xrightarrow{\bar{w}} s'} \max(0, -\bar{w}[j]) \right)$. To see this we note that between each subsequent visit to any state we only need to

visit all other states at most once (otherwise we could remove a loop or Player 2 could force an arbitrary low accumulated weight), and thus we subtract at most M in each coordinate between subsequent visits. Setting $e(p_i) = M$ for all $i \in \{1, \ldots, \ell\}$ we can apply the decidability results from [9] on the game G_e (either with or without $\bar{b}$ as weak upper bound). Thus p_{W_T} and p_{I_T} can be answered. □

Using the algorithm presented in [10], this leads to the following corollary.

Corollary 2. *The sets $min(W_T)$ and $min(I_T)$ are computable.*

In the case of GLU with parametrised transitions and $\bar{b}$ as the upper bound the set U_T is not upward closed. However, the set of useful evaluations is finite, since any winning evaluation must satisfy $-\bar{b}[i] \leq e(t)[i] \leq \bar{b}[i]$ for all transitions t and all $i \in \{1, \ldots, k\}$. This set $min(U_T)$ can therefore be constructed by an exhaustive search of the possible winning evaluations (again using the decidability results from [9]).

6 Conclusion and Future Work

Using the algorithm of Valk and Jantzen [10] we have shown how to characterise the set of winning upper bounds for multiweighted energy games with fixed initial value and a weak upper bound. For a strict upper bound the problem is solvable using $2k$-exponential time. Furthermore we have studied multiweighted energy games with parametrised transitions. For a fixed initial value and either a weak upper bound or no upper bound the same algorithm is applied to construct the set of winning evaluations. For a strict upper bound the set is shown computable as well.

Future work should include an investigation of the complexity of the above problems. As there is no upper bound on the complexity of the Valk and Jantzen algorithm, we have so far no complexity results for the results relying on the algorithm. Another future work regards parametrised transitions, where we so far are able to synthesise only nonnegative weights, and thus require that weights known to be negative are not parametrised. A likely expansion is therefore to synthesising the set of winning evaluations for energy games where any weight coordinate can be unknown. Furthermore the subject of simultaneous synthesis should be explored, where we consider games with combinations of parametrised values, this may be initial value, upper bound (strict or weak) or weight coordinates of transitions. Another direction of research is to study the problems in connection with imperfect information.

References

1. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N.: Timed Automata with Observers under Energy Constraints. In: 13th ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2010), pp. 61–70. ACM (2010)

2. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Srba, J.: Infinite Runs in Weighted Timed Automata with Energy Constraints. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 33–47. Springer, Heidelberg (2008)
3. Brázdil, T., Jančar, P., Kučera, A.: Reachability Games on Extended Vector Addition Systems with States. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010, Part II. LNCS, vol. 6199, pp. 478–489. Springer, Heidelberg (2010)
4. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource Interfaces. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 117–133. Springer, Heidelberg (2003)
5. Chaloupka, J.: Z-Reachability Problem for Games on 2-Dimensional Vector Addition Systems with States Is in P. In: Kučera, A., Potapov, I. (eds.) RP 2010. LNCS, vol. 6227, pp. 104–119. Springer, Heidelberg (2010)
6. Chatterjee, K., Doyen, L.: Energy Parity Games. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010, Part II. LNCS, vol. 6199, pp. 599–610. Springer, Heidelberg (2010)
7. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.F.: Generalized Mean-payoff and Energy Games. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010). LIPIcs, vol. 8, pp. 505–516. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2010)
8. Degorre, A., Doyen, L., Gentilini, R., Raskin, J.-F., Toruńczyk, S.: Energy and Mean-Payoff Games with Imperfect Information. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 260–274. Springer, Heidelberg (2010)
9. Fahrenberg, U., Juhl, L., Larsen, K.G., Srba, J.: Energy Games in Multiweighted Automata. In: Cerone, A., Pihlajasaari, P. (eds.) ICTAC 2011. LNCS, vol. 6916, pp. 95–115. Springer, Heidelberg (2011)
10. Valk, R., Jantzen, M.: The residue of vector sets with applications to decidability problems in Petri nets. In: Rozenberg, G. (ed.) APN 1984. LNCS, vol. 188, pp. 234–258. Springer, Heidelberg (1985)

On the Relationship between LTL Normal Forms and Büchi Automata

Jianwen Li[1], Geguang Pu[1,⋆], Lijun Zhang[2,3],
Zheng Wang[1], Jifeng He[1], and Kim Guldstrand Larsen[4]

[1] Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, P.R. China
[2] State Key Laboratory of Computer Science, Institute of Software,
Chinese Academy of Sciences
[3] Technical University of Denmark, DTU Compute, Denmark
[4] Computer Science, Aalborg University, Denmark

Abstract. In this paper, we revisit the problem of translating LTL formulas to Büchi automata. We first translate the given LTL formula into a special *disjuctive-normal form* (DNF). The formula will be part of the state, and its DNF normal form specifies the atomic properties that should hold immediately (labels of the transitions) and the *formula* that should hold afterwards (the corresponding successor state). If the given formula is Until-free or Release-free, the Büchi automaton can be obtained directly in this manner. For a general formula, the construction is more involved: an additional component will be needed for each formula that helps us to identify the set of accepting states. Notably, our construction is an on-the-fly construction, which starts with the given formula and explores successor states according to the normal forms. We implement our construction and compare the tool with SPOT [3]. The comparision results are very encouraging and show our construction is quite innovative.

1 Introduction

Translating Linear Temporal Logic (LTL) formulas to their equivalent automata (usually Büchi automata) has been studied for nearly thirty years. This translation plays a key role in the automata-based model checking [14]: here the automaton of the negation of the LTL property is first constructed, then the verification process is reduced to the emptiness problem of the product automaton (from the property automaton and the system model). Gerth et al. [6] proposed an on-the-fly construction approach to generating Büchi automata from LTL formulas, in which counterexamples can be detected even only a part of the property automaton is generated. They called it a tableau construction approach, which became widely used and many subsequent works [11,7,2,4,1] for optimizing the automata under construction are based on it.

In this paper, we propose a novel construction by making use of the notion of *disjuctive-normal forms* (DNF). For an LTL formula φ, our DNF normal form is an equivalent formula of the form $\bigvee_i(\alpha_i \wedge X\varphi_i)$ where α_i is a finite conjunction of literals (atomic propositions or their negations), and φ_i is a conjunctive LTL formula such

⋆ Corresponding author.

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 256–270, 2013.

that the root operator of it is not a disjunction. We show that any LTL formula can be transformed into an equivalent DNF normal form, and refer to $\alpha_i \wedge X\varphi_i$ as a clause of φ. In this way any given LTL formula induces a labelled transition system (LTS): states correspond to formulas, and we assign a transition from φ to φ_i labelled with α_i, if $\alpha_i \wedge X\varphi_i$ is a clause of φ.

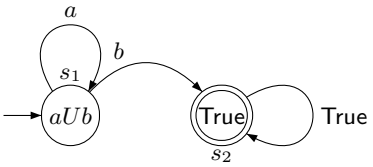


Fig. 1. The Büchi automaton for aUb

This LTS is the starting point of our construction. Firsly, for Until-free (or Release-free) formulas, the Büchi automaton can be obtained directly by equipping the above LTS with the set of accepting states, which is illustrated as follows. Consider the formula aUb, whose DNF form is $(b \wedge X(\mathsf{True})) \vee (a \wedge X(aUb))$. The corresponding Büchi automaton for aUb is shown in Figure 1 where nodes aUb and True represent formulas aUb and True respectively. The transitions are self-explained. By semantics, we know that if the run ξ satisfies a Release-free formula φ, then there must be a finite satisfying prefix η of ξ such that any paths starting with η satisfy φ as well. Thus, for this class of formulas, the state corresponding to the formula True is considered as the single accepting state. The Until-free formulas can be treated in a similar way by taking the set of all states as accepting.

The main contribution of the paper is to extend the above construction to general formulas. As an example we consider the formula $\psi = G(aUb)$, which has the normal form $(b \wedge X\psi) \vee (a \wedge X(aUb \wedge \psi))$. Note here the formula True will be even not reachable. The most challenging part of the construction will then be identification of the set of accepting states. For this purpose, we identify first subformulas that will be reached infinitely often, which we call looping formulas. Only some of the looping formulas contribute to the set of accepting states. These formulas will be the key to our construction: we characterize a set of atomic propositions for each formula, referred to as the *obligation set*. The set contains various obligations, each represented as a set of literals, that must occur infinitely often to make the given formula satisfiable. In our construction, we add an additional component to the states to keep track of the obligations, and then define accepting states based on it – an illustrating example can be found in Section 2.

Our construction for general formula works on-the-fly: it starts with the given formula and explore successor states according to the normal forms. We implement our construction and compare the tool with SPOT [3]. The results show our tool competes with, and sometimes outperforms SPOT under the benchmarks tested, which is encouraging as SPOT is the state-of-the-art tool that has been highly optimized.

Related Work. As we know, there are two main approaches of Büchi automata construction from LTL formulas. The first approach generates the alternating automaton from the LTL formula and then translates it to the equivalent Büchi automaton [13]. Gastin et al. [5] proposed a variant of this construction in 2001, which first translates the very weak alternating co-Büchi automaton to generalised automaton with accepting transitions which is then translated into Büchi automaton. In particular, the experiments show that their algorithm outperforms the others if the formulas under construction are restricted on fairness conditions. Recently Babiak et al. [1] proposed some optimization strategies based on the work [5].

The second approach was proposed in 1995 by Gerth et al. [6], which is called the *tableau* construction. This approach can generate the automata from LTL on-the-fly, which is widely used in the verification tools for acceleration of the automata-based verification process. Introducing the (state-based) *Generalized Büchi Automata* (GBA) is the important feature for the tableau construction. Daniele et al. [2] improved the tableau construction by some simple syntactic techniques. Giannakopoulou and Lerda [7] proposed another construction approach that uses the transition-based Generalized Büchi automaton (TGBA). Some optimization techniques [4,11] have been proposed to reduce the size of the generated automata. For instance, Etessami and Holzmann [4] described the optimization techniques including proof theoretic reductions (formulas rewritten), core algorithm tightening and the automata theoretic reductions (simulation based).

Organization of the paper. Section 2 illustrates our approach by a running example. Section 3 introduces preliminaries of Büchi automata and LTL formulas; Section 4 specifies the proposed *DNF-based* construction; Section 5 compares the experimental results from our tool and SPOT. Section 6 discusses how our approach is related to the tableau construction. Section 7 concludes the paper. Proofs can be found in the report [8].

2 A Running Example

We consider the formula $\varphi_1 = \mathrm{G}(bUc \wedge dUe)$ as our running example. The DNF form of φ_1 is given by:

$$\varphi_1 = (c \wedge e \wedge X(\varphi_1)) \vee (b \wedge e \wedge X(\varphi_2)) \vee (c \wedge d \wedge X(\varphi_3)) \vee (b \wedge d \wedge X(\varphi_4)$$

where $\varphi_2 = bUc \wedge G(bUc \wedge dUe)$, $\varphi_3 = dUe \wedge G(bUc \wedge dUe)$, $\varphi_4 = bUc \wedge dUe \wedge G(bUc \wedge dUe)$. It is easy to check that the above DNF form is indeed equivalent to formula φ_1. Interestingly, we note that $\varphi_1, \varphi_2, \varphi_3, \varphi_4$ all have the same DNF form above.

The corresponding Büchi automaton for φ_1 is depicted in Fig. 2. We can see that there are four states in the generated automata, corresponding to the four formulas $\varphi_i (i = 1, 2, 3, 4)$. The state corresponding to the formula φ_1 is also the initial state. The transition relation is obtained by observing the DNF forms: for instance we have a self-loop for state s_1 with label $c \wedge e$. If we observe the normal form of φ_1, we can see that there is a term $(c \wedge e \wedge X(\varphi_1))$, where there is a conjunction of two terms $c \wedge e$ and

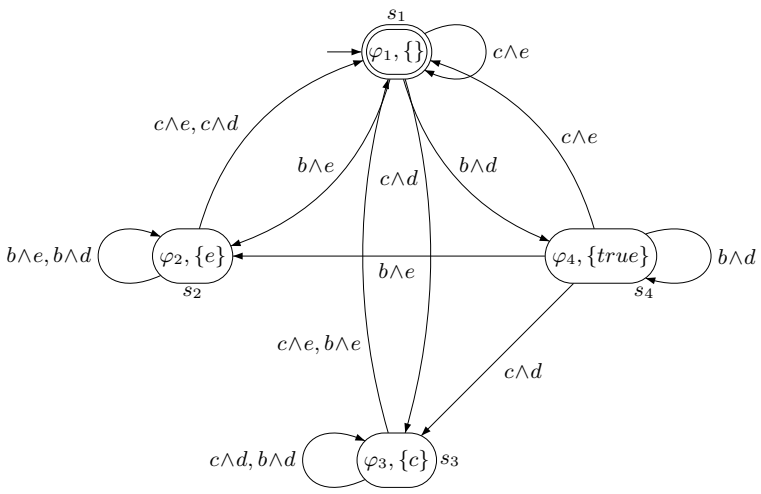


Fig. 2. The Büchi automaton for the formula φ_1

$X(\varphi_1)$, and φ_1 in X operator corresponds to the node s_1 and $c \wedge e$ corresponds to the loop edge for s_1.

Thus, the *disjunctive-normal form* of the formula has a very close relation with the generated automaton. The most difficult part is to determine the set of accepting states of the automaton. We give thus here a brief description of several notions introduced for this purpose in our running example. The four of all the formulas $\varphi_i (i = 1, 2, 3, 4)$ have the same *obligation set*, i.e. $OS_{\varphi_i} = \{\{c, e\}\}$, which may vary for different formulas. In our construction, every *obligation* in the *obligation set* of each formula identities the properties needed to be satisfied infinitely if the formula is satisfiable. For example, the formulas $\varphi_i (i = 1, 2, 3, 4)$ are satisfied if and only if all properties in the obligation $\{c, e\}$ are met infinitely according to our framework. Then, a state consists of a formula and the *process set*, which records all the properties that have been met so far. For simplicity, we initialize the *process set* P_1 of the initial state s_1 with the empty set. For the state s_2, the corresponding process set $P_2 = \{e\}$ is obtained by taking the union of P_1 and the label $\{b, e\}$ from s_1. The label b will be omitted as it is not contained in the obligation. Similarly one can conclude $P_3 = \{c\}$ and $P_4 = \{true\}$: here the property $true$ implies no property has been met so far. When there is more than one property in the *process set*, the $\{true\}$ can be erased, such as that in state s_3. Moreover, the *process set* in a state will be reset to empty if it includes one *obligation* in the formula's *obligation set*. For instance, the transition in the figure $s_2 \xrightarrow{c \wedge d} s_1$ is due to that $P_1' = P_2 \cup \{c\} = \{c, e\}$, which is actually in OS_{φ_1}. So P_1' is reset to the empty set. One can also see the same rule when the transitions $s_2 \xrightarrow{c \wedge e} s_1$, $s_4 \xrightarrow{c \wedge e} s_1$, $s_3 \xrightarrow{b \wedge e} s_1$ occur.

Through the paper, we will go back to this example again when we explain our construction approach.

3 Büchi Automaton, LTL Formulas

3.1 Büchi Automaton

A Büchi automaton is a tuple $\mathcal{A} = (S, \Sigma, \delta, S_0, F)$, where S is a finite set of states, Σ is a finite set of alphabet symbols , $\delta : S \times \Sigma \to 2^S$ is the transition relation, S_0 is a set of initial states, and $F \subseteq S$ is a set of accepting states of $\mathcal{A}$.

We use $w, w_0 \in \Sigma$ to denote alphabets in Σ, and $\eta, \eta_0 \in \Sigma^*$ to denote finite sequences. A *run* $\xi = w_0 w_1 w_2 \ldots$ is an infinite sequence over Σ^ω. For ξ and $k \geq 1$ we use $\xi^k = w_0 w_1 \ldots w_{k-1}$ to denote the prefix of ξ up to its kth element (the $k+1$th element is not included) as well as ξ_k to denote the suffix of $w_k w_{k+1} \ldots$ from its $(k+1)$th element (the $k+1$th element is included). Thus, $\xi = \xi^k \xi_k$. For notational convenience we write $\xi_0 = \xi$ and $\xi^0 = \varepsilon$ (ε is the empty string). The run ξ is accepting if it runs across one of the states in F infinitely often.

3.2 Linear Temporal Logic

We recall the linear temporal logic (LTL) which is widely used as a specification language to describe the properties of reactive systems. Assume AP is a set of atomic properties, then the syntax of LTL formulas is defined by:

$$\varphi \ ::= \mathsf{True} \mid \mathsf{False} \mid \ a \mid \neg a \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \, U \varphi \mid \varphi \, R \, \varphi \mid X \, \varphi$$

where $a \in AP$, φ is an LTL formula. We say φ is a *literal* if it is a proposition or its negation. In this paper we use lower case letters to denote atomic properties and α, β, γ to denote propositional formulas (without temporal operators), and use φ, ψ, ϑ, μ, ν and λ to denote LTL formulas.

Note that w.l.o.g. we are considering LTL formulas in negative normal form (NNF) – all negations are pushed down to literal level. LTL formulas are interpreted on infinite sequences (correspond to runs of the automata) $\xi \in \Sigma^\omega$ with $\Sigma = 2^{AP}$. The Boolean connective case is trivial, and the semantics of temporal operators is given by:

- $\xi \models \varphi_1 \, U \, \varphi_2$ iff there exists $i \geqslant 0$ such that $\xi_i \models \varphi_2$ and for all $0 \leqslant j < i, \xi_j \models \varphi_1$;
- $\xi \models \varphi_1 \, R \, \varphi_2$ iff either $\xi_i \models \varphi_2$ for all $i \geq 0$, or there exists $i \geq 0$ with $\xi_i \models \varphi_1 \wedge \varphi_2$ and $\xi_j \models \varphi_2$ for all $0 \leq j < i$;
- $\xi \models X \, \varphi$ iff $\xi_i \models \varphi$.

According to the LTL semantics, it holds $\varphi R \psi = \neg(\neg\varphi U \neg\varphi)$. We use the usual abbreviations $\mathsf{True} = a \vee \neg a$, $Fa = \mathsf{True} U a$ and $Ga = \mathsf{False} R a$.

Notations. Let φ be a formula written in *conjunctive form* $\varphi = \bigwedge_{i \in I} \varphi_i$ such that the root operator of φ_i is not a conjunctive: then we define the conjunctive formula set as $CF(\varphi) := \{\varphi_i \mid i \in I\}$. When φ does not include a conjunctive as a root operator, $CF(\varphi)$ only includes φ itself. For technical reasons, we assume that $CF(\mathsf{True}) = \emptyset$.

4 *DNF-Based* Büchi Automaton Construction

Our goal of this section is to construct the Büchi automaton $\mathcal{A}_\lambda$ for λ. We first present the disjunctive normal forms, and then establish a few simple properties of general formulas that shall shed insights on the construction for the $Release$-free ($Until$-free) formulas. We then define the labelled transition system for a formula. In the following three subsections we present the construction for $Release$-free ($Until$-free) and general formulas, respectively.

In the remaining of the paper, we fix λ as the input LTL formula. All formulas being considered will vary over the set $EF(\lambda)$, and AP will denote the set of all literals appearing in λ, and $\Sigma = 2^{AP}$. Moreover, we assume all atomic propositions in λ are *syntactically different*, but maintain their *semantically equivalence*. For example, the formula $aU(a \wedge b)$ will be identified by $a_1U(a_2 \wedge b)$, where $a_1 \equiv a_2$. This will be used to track whether the atomic proposition are from the left part or right part of the until operator.

4.1 Disjunctive Normal Form

We introduce the notion of *disjunctive-normal form* for LTL formulas in the following.

Definition 1 (disjunctive-normal form). *A formula φ is in disjunctive-normal form (DNF) if it can be represented as $\varphi := \bigvee_i(\alpha_i \wedge X\varphi_i)$, where α_i is a finite conjunction of literals, and $\varphi_i = \bigwedge \varphi_{i_j}$ where φ_{i_j} is either a literal, or an Until, Next or Release formula.*

We say $\alpha_i \wedge X\varphi_i$ is a clause *of φ, and write $DNF(\varphi)$ to denote all of the clauses.*

As seen in the introduction and motivating example, DNF form plays a central role in our construction. Thus, we first discuss that any LTL formula φ can be transformed into an equivalent formula in DNF form. The transformation is done in two steps. The first step is according to the following rules:

Lemma 1. *1.* $DNF(\alpha) = \{\alpha \wedge X(\mathsf{True})\}$ *where α is a literal;*
2. $DNF(X\varphi) = \{\mathsf{True} \wedge X(\varphi)\}$*;*
3. $DNF(\varphi_1 U \varphi_2) = DNF(\varphi_2) \cup DNF(\varphi_1 \wedge X(\varphi_1 U \varphi_2))$*;*
4. $DNF(\varphi_1 R \varphi_2) = DNF(\varphi_1 \wedge \varphi_2) \cup DNF(\varphi_2 \wedge X(\varphi_1 R \varphi_2))$*;*
5. $DNF(\varphi_1 \vee \varphi_2) = DNF(\varphi_1) \cup DNF(\varphi_2)$*;*
6. $DNF(\varphi_1 \wedge \varphi_2) = \{(\alpha_1 \wedge \alpha_2) \wedge X(\psi_1 \wedge \psi_2) \mid \forall i = 1, 2.\, \alpha_i \wedge X(\psi_i) \in DNF(\varphi_i)\}$*;*

All of the rules above are self explained, following by the definition of DNF, distributive and the expansion laws. What remains is how to deal with the formulas in the *Next* operator: by definition, in a clause $\alpha_i \wedge X(\varphi_i)$ the root operators in φ_i cannot be disjunctions. The equivalence $X(\varphi_1 \vee \varphi_2) = X\varphi_1 \vee X\varphi_2$ can be applied repeatedly to move the disjunctions out of the *Next* operator. The distributive law of disjunction over conjunctions allows us to bring any formula into an equivalent DNF form:

Theorem 1. *Any LTL formula φ can be transformed into an equivalent formula in disjunctive-normal form.*

In our running example, we have $DNF(\varphi_1) = DNF(\varphi_2) = DNF(\varphi_3) = DNF(\varphi_4) = \{c \wedge e \wedge X(\varphi_1), b \wedge e \wedge X(\varphi_2), c \wedge d \wedge X(\varphi_3), b \wedge d \wedge X(\varphi_4)\}$. Below we discuss the set of formulas that can be reached from a given formula.

Definition 2 (Formula Expansion). *We write $\varphi \xrightarrow{\alpha} \psi$ iff there exists $\alpha \wedge X(\psi) \in DNF(\varphi)$. We say ψ is expandable from φ, written as $\varphi \hookrightarrow \psi$, if there exists a finite expansion $\varphi \xrightarrow{\alpha_1} \psi_1 \xrightarrow{\alpha_2} \psi_2 \xrightarrow{\alpha_3} \ldots \psi_n = \psi$. Let $EF(\varphi)$ denote the set of all formulas that can be expanded from φ.*

The following theorem points out that $|EF(\lambda)|$ is bounded:

Theorem 2. *For any formula λ, $|EF(\lambda)| \leq 2^n + 1$ where n denotes the number of subformulas of λ.*

4.2 Transition Systems for LTL Formulas

We first extend formula expansions to subset in Σ:

Definition 3. *For $\omega \in \Sigma$ and propositional formula α, $\omega \models \alpha$ is defined in the standard way: if α is a literal, $\omega \models \alpha$ iff $\alpha \in \omega$, and $\omega \models \alpha_1 \wedge \alpha_2$ iff $\omega \models \alpha_1$ and $\omega \models \alpha_2$, and $\omega \models \alpha_1 \vee \alpha_2$ iff $\omega \models \alpha_1$ or $\omega \models \alpha_2$.*

We write $\varphi \xrightarrow{\omega} \psi$ if $\varphi \xrightarrow{\alpha} \psi$ and $w \models \alpha$. For a word $\eta = \omega_0\omega_1..\omega_k$, we write $\varphi \xrightarrow{\eta} \psi$ iff $\varphi \xrightarrow{\omega_0} \psi_1 \xrightarrow{\omega_1} \psi_2 \xrightarrow{\omega_2} ..\psi_{k+1} = \psi$.

For a run $\xi \in \Sigma^\omega$, we write $\varphi \xrightarrow{\xi} \varphi$ iff ξ can be written as $\xi = \eta_0\eta_1\eta_2 \ldots$ such that η_i is a finite sequence, and $\varphi \xrightarrow{\eta_i} \varphi$ for all $i \geq 0$.

Below we provide a few interesting properties derived from our DNF normal forms.

Lemma 2. *Let ξ be a run and λ a formula. Then, for all $n \geq 1$, $\xi \models \lambda \Leftrightarrow \lambda \xrightarrow{\xi^n} \varphi \wedge \xi_n \models \varphi$.*

Essentially, $\xi \models \lambda$ is equivalent to that we can reach a formula φ along the prefix ξ^n such that the suffix ξ_n satisfies φ. The following corollary is a direct consequence of Lemma 2 and the fact that we have only finitely many formulas in $EF(\lambda)$:

Corollary 1. *If $\xi \models \lambda$, then there exists $n \geq 1$ such that $\lambda \xrightarrow{\xi^n} \varphi \wedge \xi_n \models \varphi \wedge \varphi \xrightarrow{\xi_n} \varphi$. On the other side, if $\lambda \xrightarrow{\xi^n} \varphi \wedge \xi_n \models \varphi \wedge \varphi \xrightarrow{\xi_n} \varphi$, then $\xi \models \lambda$.*

This corollary gives the hint that after a finite prefix we can focus on whether the suffix satisfies the *looping formula* φ, i.e,. those φ with $\varphi \hookrightarrow \varphi$. From Definition 2 and the expansion rules for LTL formulas, we have the following corollary:

Corollary 2. *If $\lambda \hookrightarrow \lambda$ holds and $\lambda \neq$ True, then there is at least one Until or Release formula in $CF(\lambda)$.*

As we described in previous, the elements in $EF(\lambda)$ and its corresponding DNF-normal forms naturally induce a labelled transition system, which can be defined as follows:

Definition 4 (LTS for λ). *The labelled transition system TS_λ generated from the formula λ is a tuple $\langle \Sigma, S, \delta, S_0 \rangle$: where $\Sigma = 2^{AP}$, $S = EF(\lambda)$, $S_0 = \{\lambda\}$ and δ is defined as follows: $\psi \in \delta(\varphi, \omega)$ iff $\varphi \xrightarrow{\omega} \psi$ holds, where $\varphi, \psi \in EF(\lambda)$ and $\omega \in \Sigma$.*

4.3 Büchi Automata for Release/Until-Free Formulas

The following lemma is a special instance of our central theorem 4. It states properties of accepting runs with respect to Release/Until-free formulas:

Lemma 3. *1. Assume λ is $Release$-free. Then, $\xi \models \lambda \Leftrightarrow \exists n \cdot \lambda \xrightarrow{\xi^n} \mathsf{True}$.*
2. Assume λ is $Until$-free. Then $\xi \models \lambda \Leftrightarrow \exists n, \varphi \cdot \lambda \xrightarrow{\xi^n} \varphi \wedge \varphi \xrightarrow{\xi_n} \varphi$.

Essentially, If λ is Release-free, we will reach True after finitely many steps; If λ is Until-free we will reach a looping formula after finitely many steps. The Büchi automaton for Release-free or Until-free formulas will be directly obtained by equipping the LTS with the set of accepting states:

Definition 5 ($\mathcal{A}_\lambda$ for Release/Until-free formulas). *For a Release/Until-free formula λ, we define the Büchi automaton $\mathcal{A}_\lambda = (S, \Sigma, \delta, S_0, F)$ where $TS_\lambda = \langle \Sigma, S, \delta, S_0 \rangle$. The set F is defined by: $F = \{\mathsf{True}\}$ if λ is Release-free while $F = S$ if λ is Until-free.*

Notably, True is the only accepting state for $\mathcal{A}_\lambda$ when λ is Release-free while all the states are accepting ones if it is Until-free.

Theorem 3 (Correctness and Complexity). *Assume λ is $Until$-free or $Release$-free. Then, for any sequence $\xi \in \Sigma^\omega$, it holds $\xi \models \lambda$ iff ξ is accepted by $\mathcal{A}_\lambda$. Moreover, $\mathcal{A}_\lambda$ has at most $2^n + 1$ states, where n is the number of subformulas in λ.*

Proof. The proof of the correctness is trivial according to Lemma 3: 1) if λ is Release-free, then every run ξ of $\mathcal{A}_\lambda$ can run across the True-state[1] infinitely often iff it satisfies $\exists n \geq 0 \cdot \lambda \xrightarrow{\xi^n} \mathsf{True}$, that is, $\xi \models \lambda$; 2) if λ is Until-free, then $\xi \models \lambda$ iff $\exists n, \varphi \cdot \lambda \xrightarrow{\xi^n} \varphi \wedge \varphi \xrightarrow{\xi_n} \varphi$, which will run across φ-state infinitely often so that is accepted by $\mathcal{A}_\lambda$ according to the construction.

The upper bound is a direct consequence of Theorem 2.

4.4 Central Theorem for General Formulas

In the previous section we have constructed Büchi automaton for Release-free or Until-free formulas, which is obtained by equipping the defined LTS with appropriate accepting states. For general formulas, this is however slightly involved. For instance, consider the LTS of the formula $\varphi = G(bUc \wedge dUe)$ in our running example: there are infinitely many runs starting from the initial state s_1, but which of them should be accepting? Indeed, it is not obvious how to identify the set of accepting states. In this section we present our central theorem for general formulas aiming at identifying the accepting runs.

Assume the run $\xi = \omega_0\omega_1 \ldots$ satisfies the formula λ. We refer to $\lambda (= \varphi_0) \xrightarrow{\alpha_0} \varphi_1 \xrightarrow{\alpha_1} \varphi_2 \ldots$ as an expansion path from λ, which corresponds to a path in the LTS TS_λ, but labelled with propositional formulas. Obviously, $\xi \models \lambda$ implies that there exists an expansion path in TS_λ such that $\omega_i \models \alpha_i$ for all $i \geq 0$. As the set $EF(\lambda)$ is

[1] In this paper we use φ-state to denote the state representing the formula φ.

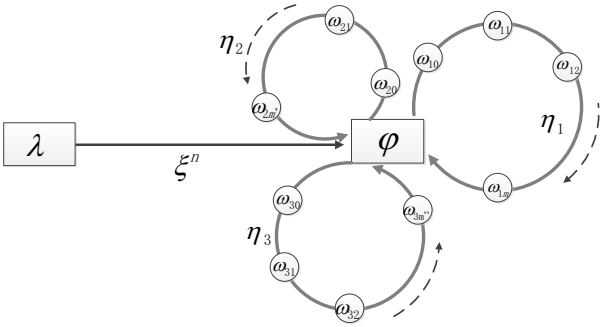


Fig. 3. A snapshot illustrating the relation $\xi \models \lambda$

finite, we can find a looping formula $\varphi = \varphi_i$ that occurs *infinitely often* along this expansion path. On the other side, we can *partition* the run ξ into sequences $\xi = \eta_0\eta_1 \ldots$ such each finite sequence η_i is consistent with respect to one loop $\varphi \hookrightarrow \varphi$ along the expansion path. This is illustrated in Figure 3. The definition below formalizes the notion of consistency for finite sequence:

Definition 6. *Let $\eta = \omega_0\omega_1 \ldots \omega_n$ ($n \geq 0$) be a finite sequence. Then, we say that η satisfies the LTL formula φ, denoted by $\eta \models_f \varphi$, if the following conditions are satisfied:*

- *there exists $\varphi_0 = \varphi \xrightarrow{\alpha_0} \varphi_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \varphi_{n+1} = \psi$ such that $\omega_i \models \alpha_i$ for $0 \leq i \leq n$, and with $S := \bigcup_{0 \leq j \leq n} CF(\alpha_j)$, it holds*
  1. *if φ is a literal then $\varphi \in S$ holds;*
  2. *if φ is $\varphi_1 U \varphi_2$ or $\varphi_1 R \varphi_2$ then $S \models_f \varphi_2$ holds;*
  3. *if φ is $\varphi_1 \wedge \varphi_2$ then $S \models_f \varphi_1 \wedge S \models_f \varphi_2$ holds;*
  4. *if φ is $\varphi_1 \vee \varphi_2$ then $S \models_f \varphi_1 \vee S \models_f \varphi_2$ holds;*
  5. *if φ is $X\varphi_2$ then $S \models_f \varphi_2$ holds;*

This predicate specifies whether the given finite sequence η is consistent with respect to the finite expansion $\varphi_0 = \varphi \xrightarrow{\alpha_0} \varphi_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \varphi_{n+1} = \psi$. The condition $\omega_i \models \alpha_i$ requires that the finite sequence η is consistent with respect to the labels along the finite expansion from φ_0. The rules for literals and Boolean connections are intuitive. For Until operator $\varphi_1 U \varphi_2$, it is defined recursively by $S \models_f \varphi_2$: as to make the Until subformula being satisfied, we should make sure that φ_2 holds under S. Similar, for release operator $\varphi_1 R \varphi_2$, we know that $\varphi_1 \wedge \varphi_2$ or φ_2 plays a key role in an accepting run of $\varphi_1 R \varphi_2$. Because $\varphi_1 \wedge \varphi_2$ implies φ_2, and with the rule (4) in the definition, we have $S \models_f \varphi_1 R \varphi_2 \equiv S \models_f \varphi_2$. Assume $\varphi = X\varphi_2$. As $CF(\mathsf{True})$ is defined as $\emptyset$, we have $\eta \models_f \varphi$ iff $\eta' \models_f \varphi_2$ with $\eta' = \omega_1\omega_2 \ldots \omega_n$.

The predicate $\models_f$ characterizes whether the prefix of an accepting run contributes to the satisfiability of λ. The idea comes from Corollary 1: Once φ is expanded from itself infinitely by a run ξ as well as $\xi \models \varphi$, there must be some common feature each time φ loops back to itself. This common feature is what we defined in $\models_f$. In our running example, consider the finite sequence $\eta = \{b, d\}\{b, d\}\{c, e\}$ corresponding to the path

$s_1 s_4 s_4 s_1$: according to the definition $\eta \models_f \varphi_1$ holds. For $\eta = \{b,d\}\{b,d\}\{b,d\}$, however, $\eta \not\models_f \varphi_1$.

With the notation $\models_f$, we study below properties for the looping formulas, that will lead to our *central theorem*.

Lemma 4 (Soundness). *Given a looping formula φ and an infinite word ξ, let $\xi = \eta_1\eta_2 \ldots$. If $\forall i \geq 1 \cdot \varphi \xrightarrow{\eta_i} \varphi \wedge \eta_i \models_f \varphi$, then $\xi \models \varphi$.*

The soundness property of the looping formula says that if there exists a partitioning $\xi = \eta_1\eta_2 \ldots$ such that φ expends to itself by each η_i and $\eta_i \models_f \varphi$ holds, then $\xi \models \varphi$.

Lemma 5 (Completeness). *Given a looping formula φ and an infinite word ξ, if $\varphi \xrightarrow{\xi} \varphi$ and $\xi \models \varphi$ holds, then there exists a partitioning $\eta_1\eta_2 \ldots$ for ξ, i.e. $\xi = \eta_1\eta_2 \ldots$, such that for all $i \geq 0$, $\varphi \xrightarrow{\eta_i} \varphi \wedge \eta_i \models_f \varphi$ holds.*

The completeness property of the looping formula states the other direction. If $\varphi \xrightarrow{\xi} \varphi$ as well as $\xi \models \varphi$, we can find a partitioning $\eta_1\eta_2 \ldots$ that makes φ expending to itself by each η_i and $\eta_i \models_f \varphi$ holds. Combining Lemma 6, Lemma 7 and Corollary 1, we have our central theorem:

Theorem 4 (Central Theorem). *Given a formula λ and an infinite word ξ, we have*

$$\xi \models \lambda \Leftrightarrow \exists \varphi, n \cdot \lambda \xrightarrow{\xi^n} \varphi \wedge \exists \xi_n = \eta_1\eta_2 \ldots \cdot \forall i \geq 1 \cdot \varphi \xrightarrow{\eta_i} \varphi \wedge \eta_i \models_f \varphi$$

The central theorem states that given a formula λ, we can always extend it to a looping formula which satisfies the soundness and completeness properties. Reconsider Figure 3: formula λ extends to the looping formula φ by ξ^n, and ξ_n can be partitioned into sequences $\eta_1\eta_2 \ldots$. The loops from φ correspond to these finite sequences η_i in the sense $\eta_i \models_f \varphi$.

4.5 Büchi Automata for General Formulas

Our central theorem sheds insights about the correspondence between the accepting run and the expansion path from λ. However, how can we guarantee the predicate $\models_f$ for looping formulas in the theorem? We need the last ingredient for starting our automaton construction: we extract the *obligation sets* from LTL formulas that will enable us to characterize $\models_f$.

Definition 7. *Given a formula φ, we define its obligation set, i.e. OS_φ, as follows:*

1. *If $\varphi = \mathsf{True}$ then $OS_\varphi = \{\emptyset\}$; and if $\varphi = \mathsf{False}$ then $OS_\varphi = \{\{\mathsf{False}\}\}$;*
2. *If $\varphi = p$, $OS_\varphi = \{\{p\}\}$;*
3. *If $\varphi = X\psi$, $OS_\varphi = OS_\psi$;*
4. *If $\varphi = \psi_1 \vee \psi_2$, $OS_\varphi = OS_{\psi_1} \cup OS_{\psi_2}$;*
5. *If $\varphi = \psi_1 \wedge \psi_2$, $OS_\varphi = \{S_1 \cup S_2 \mid S_1 \in OS_{\psi_1} \wedge S_2 \in OS_{\psi_2}\}$;*
6. *If $\varphi = \psi_1 U \psi_2$ or $\psi_1 R \psi_2$, $OS_\varphi = OS_{\psi_2}$;*

For every element set $O \in OS_\varphi$, we call it the obligation of φ.

The obligation set provides all obligations (elements in obligation set) the given formula is supposed to have. Intuitively, a run ξ is accepted a formula φ if ξ can eliminate the obligations of φ. Take the example of $G(aRb)$, the run $(b)^\omega$ is accepted by aRb, and the run eliminates the obligation $\{b\}$ infinitely often.

Notice the similarity of the definition of the obligation set and the predicate $\models_f$. For instance, the obligation set of $\varphi_1 R \varphi_2$ is the obligation set of φ_2, which is similar in the definition of $\models_f$. The interesting rule is the conjunctive one. For obligation set OS_φ, there may be more than one element in OS_φ. However, from the view of satisfiability, if one obligation in OS_φ is satisfied, we can say the obligations of φ is fulfilled. This view leads to the definition of the conjunctive rule. For $\psi_1 \wedge \psi_2$, we need to fulfill the obligations from both ψ_1 and ψ_2, which means we have to trace all possible unions from the elements of OS_{ψ_1} and OS_{ψ_2}. For instance, the obligation set of $G(aUb \wedge cU(d \vee e))$ is $\{\{b, d\}, \{b, e\}\}$. The following lemmas gives the relationship of $\models_f$ and *obligation set*.

Lemma 6. *For all $O \in OS_\varphi$, it holds $O \models_f \varphi$. On the other side, $S \models_f \varphi$ implies that $\exists O \in OS_\varphi \cdot O \subseteq S$.*

For our input formula λ, now we discuss how to construct the Büchi automaton $\mathcal{A}_\lambda$. We first describe the states of the automaton. A state will be consisting of the formula φ and a *process set* that keeps track of properties have been satisfied so far. Formally:

Definition 8 (States of the automaton for λ). *A state is a tuple $\langle \varphi, P \rangle$ where φ is a formula from $EF(\lambda)$, and $P \subseteq AP$ is a process set.*

Refer again to Figure 3: reading the input finite sequence η_1, each element in the process set P_i corresponds to a property set belonging to AP, which will be used to keep track whether all elements in an obligation are met upon returning back to a φ-state. If we have $P_i = \emptyset$, we have successfully returned to the accepting states. Now we have all ingredients for constructing our Büchi automaton $\mathcal{A}_\lambda$:

Definition 9 (Büchi Automaton $\mathcal{A}_\lambda$). *The Büchi automaton for the formula λ is defined as $\mathcal{A}_\lambda = (\Sigma, S, \delta, S_0, \mathcal{F})$, where $\Sigma = 2^{AP}$ and:*

- *$S = \{\langle \varphi, P \rangle \mid \varphi \in EF(\lambda)\}$ is the set of states;*
- *$S_0 = \{\langle \lambda, \emptyset \rangle\}$ is the set of initial states;*
- *$\mathcal{F} = \{\langle \varphi, \emptyset \rangle \mid \varphi \in EF(\lambda)\}$ is the set of accepting states;*
- *Let states s_1, s_2 with $s_1 = \langle \varphi_1, P_1 \rangle$, $s_2 = \langle \varphi_2, P_2 \rangle$ and $w \subseteq 2^{AP}$. Then, $s_2 \in \delta(s_1, \omega)$ iff there exists $\varphi_1 \xrightarrow{\alpha} \varphi_2$ with $\omega \models \alpha$ such that the corresponding P_2 is updated by:*
  1. *$P_2 = \emptyset$ if $\exists O \in OS_{\varphi_2} \cdot O \subseteq P_1 \cup CF(\alpha)$,*
  2. *$P_2 = P_1 \cup CF(\alpha)$ otherwise.*

The transition is determined by the expansion relation $\varphi_1 \xrightarrow{\alpha} \varphi_2$ such that $\omega \models \alpha$. The process set P_2 is updated by $P_1 \cup CF(\alpha)$ unless there is no element set $O \in OS_{\varphi_2}$ such that $P_1 \cup CF(\alpha) \supseteq O$. In that case P_2 will be set to $\emptyset$ and the corresponding state will be recognized as an accepting one.

Now we state the correctness of our construction:

Theorem 5 (Correctness of Automata Generation). *Let λ be the input formula. Then, for any sequence $\xi \in \Sigma^\omega$, it holds $\xi \models \lambda$ iff ξ is accepted by $\mathcal{A}_\lambda$.*

The correctness follows mainly from the fact that our construction strictly adheres to our central theorem (Theorem 4).

We note that two very simple optimizations can be identified for our construction:

- If two states have the same DNF normal form and the same process set P, they are identical. Precisely, we merge states $s_1 = \langle \varphi_1, P_1 \rangle$ and $s_2 = \langle \varphi_2, P_2 \rangle$ if $DNF(\varphi_1) = DNF(\varphi_2)$, and $P_1 = P_2$;
- The elements in the process set P can be restricted into those atomic propositions appearing in OS_φ: Recall here $\varphi \in EF(\lambda)$. One can observe directly that only those properties are used for checking the *obligation* conditions, while others will not be used so that it can be omitted in the process set P.

Now we can finally explain a final detail of our running example:

Example 1. In our running example state s_1 is the accepting state of the automaton. It should be mentioned that the state $s_2 = \langle \varphi_2, \{e\} \rangle$ originally has an edge labeling $c \wedge d$ to the state $\langle \varphi_3, \emptyset \rangle$ according to our construction, which is a new state. However, this state is equivalent with $s_1 = \langle \varphi_1, \emptyset \rangle$, as φ_1 and φ_3 have the same DNF normal form. So these two states are merged. The same cases occur on state s_3 to state s_1 with the edge labeling $b \wedge e$, state s_2 to state s_2 with the edge labeling $b \wedge d$ and etc. After merging these states, we have the automaton as depicted in Figure 2.

Theorem 6 (Complexity). *Let λ be the input formula. Then the Büchi automaton $\mathcal{A}_\lambda$ has the upper bound $2^{2n} + 1$, where n is the number of subformulas in λ.*

Proof. The number of states is bounded by $(2^n + 1) \cdot 2^{|AP|}$. As the first part contains the particular True, the bound is restricted to $2^{2n} + 1$. □

Recall in our construction in the paper we assume all atomic propositions in λ are *syntactically different*, but maintain their *semantically equivalence*. Thus, n referes to the number of subformulas after *tagging* the literals appearing in λ. Details please refer to [8].

5 Experiments

In order to show the efficiency of our construction, we implement the algorithms in our tool *Aalta*[2] and compare it with the SPOT tool [3], which is a state-of-the-art LTL-to-Büchi translator. SPOT follows the tableau framework [15] but uses BDDs [16] to make the translation more efficient. We compare the results between *Aalta* and SPOT's newest 1.0.2 version when this paper is written. Since SPOT has integrated several translations and the results vary on what the user demands so we here choose its default configuration (with flags "-l -t").

[2] `http://www.lab205.org/Aalta/`

Table 1. Comparison results between SPOT and *Aalta*. In each formula group (with the same length) the first line displays the results from SPOT while the second from *Aalta*.

Formula Length	States	Transitions	Nondet-States	Nondet	Time	Product States
10	4.32	17.99	2.69	0.75	0.14	706
	3.44	18.22	1.77	0.74	0.03	538
20	23.30	146.73	4.43	0.82	0.14	4467
	6.67	56.22	2.84	0.76	0.05	1145
30	41.90	259.15	16.32	0.85	0.14	8183
	10.52	113.27	7.62	0.78	0.10	1857
40	45.76	296.05	20.26	0.83	0.06	8909
	20.55	323.20	16.84	0.80	0.27	3857
50	167.13	1161.11	69.52	0.91	0.12	33225
	43.34	744.53	36.87	0.86	2.80	8420

SPOT also integrates the "randltl"[3] and "ltlcross"[4] scripts which are used to generate random formulas required and check the correctness as well as provide statistics. As a result we can test the correctness of *Aalta* together with obtaining comparing results by using the combination of the two scripts. We set the "randltl" script to generate random formulas with 2 variables with lengths varying from 10 to 50. In the experiments *Aalta* and SPOT are run for around 10,000 formulas, and no errors are detected by the "ltlcross" script. In this paper we assume the results from SPOT are absolutely true, so the testing affirms *Aalta*'s correctness.

In practice, it is not always true that the smaller generated automata are, the better model checking performance is. Etessami et al. [4] pointed out that the model checking performance depends on the size of the product automaton (from the property automaton and the system model) rather than that of property automata. After that Sebastiani and Tonetta [10] conclude the property automata must be "as deterministic as possible". To sum up there are five criteria for evaluating the generated automata: states, transitions, non-deterministic states, non-determinism and the generation time. Among them the transitions are countered deterministically, and the non-deterministic states are those whose outedges are not deterministic. Moreover, since the "ltlcross" script completes simple checking based on the generated automata and a universal model, so we can also gather the size of product states as an important criteria. As a result, we list the comparing results on these criteria in Table 1.

The statistics in Table 1 are arranged as follows. We first set the flags of "randltl" script to fix the alphabet with two variables as well as the probabilities of appearance for temporal operators during formula generation. Then we generate the formulas with length varying from 10 to 50 — 200 formulas for each. The table shows only the average

[3] `http://spot.lip6.fr/userdoc/randltl.html`
[4] `http://spot.lip6.fr/userdoc/ltlcross.html`

results in five groups divided by the formula length. Note in each group there are two lines of results in the table, in which the first one corresponds to SPOT and the second corresponds to *Aalta*.

Generally speaking, the smaller the number in the table the better the generated automata are. So one can see that *Aalta* can perform as well as – or even better than – SPOT under the benchmarks we tested. This is very encouraging, as SPOT has been optimized and maintained for more than ten years. The statistics in the table affirm that *Aalta* has better model checking performance than SPOT under some formulas tested. Note the results when the formula length is 50, in which *Aalta* spends more time on automata generation. Thus SPOT has a better scalability, and this is also what need to be improved in *Aalta* in future.

6 Discussion

In this section, we discuss the relationship and differences between our proposed approach and the tableau construction.

Generally speaking, our approach is essentially a tableau one that is based on the expansion laws of Until and Release operators. The interesting aspect of our approach is the finding of a special normal formal with its DNF-based labeled transition system, which is closely related to the Büchi automaton under construction. The tableau approach explicitly expands the formula recursively based on the semantics of LTL formulas while the nodes of the potential automaton are split until no new node can be generated. However, our approach first studies the LTL normal forms to discover the obligations the automaton has to fulfil, and then presents a simple mapping between LTL formulas into Büchi automata.

The insight behind our approach is adopting a different view on the accepting conditions. The tableau approach focuses on the Until-operator. For instance, to decide the accepting states, the tableau approach needs to trace all the Until-subformulas and records the "eventuality" of ψ in $\varphi U \psi$, which leads to the introduction of the *Generalized Büchi Automata* (GBA) in the tableau approach. However, our approach focuses on the *looping formulas*, which potentially consist of the accepting states. Intuitively, an infinite sequence (word) will satisfy the formula λ iff λ can expand to some looping formula φ which can be satisfied by the suffix of the word removing the finite sequence arriving at φ. The key point of our approach is to introduce the static obligation set for each formula in the DNF-based labeled transition system, which indicates that an accepting run is supposed to infinitely fulfil one of the obligations in the obligation set. Thus, the obligation set gives the "invariability" for general formulas instead of the "eventuality" for $Until$-formulas. In the approach, we use a process set to record the obligation that formula φ has been satisfied from its last appearance. Then, we would decide the accepting states easily when the process set fulfills one obligation in the obligation set of φ (We reset it empty afterwards). One can also notice our approach is on-the-fly: the successors of the current state can be obtained as soon as its DNF normal form is acquired.

7 Conclusion

In this paper, we propose the *disjunctive-normal forms* for LTL formulas. Based on that, we introduce the DNF-based labeled transition system for the input formula λ

and study the relationship between the transition system and the Büchi automata for λ. The construction makes use of the notion of obligation set, and we reach a simple but on-the-fly algorithm. When the formula under construction is Release/Until-free, our construction is very straightforward in theory.

Acknowledgement. The work is supported by IDEA4CPS, MT-LAB (a VKR Centre of Excellence), Shanghai Knowledge Service Platform for Trustworthy Internet of Things No. ZF1213 and NSFC Project No.91118007. The work of Lijun Zhang was done while he was at DTU Compute, Technical University of Denmark, Denmark.

References

1. Babiak, T., Křetínský, M., Řehák, V., Strejček, J.: LTL to Büchi Automata Translation: Fast and More Deterministic. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 95–109. Springer, Heidelberg (2012)
2. Daniele, M., Giunchiglia, F., Vardi, M.Y.: Improved Automata Generation for Linear Temporal Logic. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. CAV, pp. 249–260. Springer, Heidelberg (1999)
3. Duret-Lutz, A., Poitrenaud, D.: SPOT: an extensible model checking library using transition-based generalized Büchi automata In. In: The IEEE Computer Society's 12th Annual International Symposium, pp. 76–83 (2004)
4. Etessami, K., Holzmann, G.J.: Optimizing Büchi Automata. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 153–167. Springer, Heidelberg (2000)
5. Gastin, P., Oddoux, D.: Fast LTL to Büchi Automata Translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
6. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: PSTV, pp. 3–18 (1995)
7. Giannakopoulou, D., Lerda, F.: From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 308–326. Springer, Heidelberg (2002)
8. Li, J., Pu, G., Zhang, L., Wang, Z., He, J., Larsen, K.G.: On the Relationship between LTL Normal Forms and Buechi Automata. CoRR (2012)
9. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 149–167. Springer, Heidelberg (2007)
10. Sebastiani, R., Tonetta, S.: "More Deterministic" vs. "Smaller" Büchi Automata for Efficient LTL Model Checking. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 126–140. Springer, Heidelberg (2003)
11. Somenzi, F., Bloem, R.: Efficient Büchi Automata from LTL Formulae. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 248–263. Springer, Heidelberg (2000)
12. Tauriainen, H., Heljanko, K.: Testing LTL formula translation into Büchi automata. STTT 4(1), 57–70 (2002)
13. Vardi, M.Y.: An Automata-Theoretic Approach to Linear Temporal Logic. In: Banff Higher Order Workshop, pp. 238–266 (1995)
14. Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: LICS, pp. 332–344 (1986)
15. Couvreur, J.-M.: On-the-fly verification of linear temporal logic. In: Wing, J.M., Woodcock, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (1999)
16. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers, 677–691 (1986)

Managing Environment and Adaptation Risks for the Internetware Paradigm*

Jian Lü, Yu Huang, Chang Xu, and Xiaoxing Ma

State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
Institute of Computer Software, Nanjing University
Nanjing 210023, China
{lj,yuhuang,changxu,xxm}@nju.edu.cn

Abstract. Internetware is receiving increasing attention. It envisions a new, yet promising software engineering paradigm for constructing complex systems that are situated in open and dynamic networked environments. Typical examples of Internetware systems include Internet-based and cyber-physical systems. These systems, although having addressed some practical needs, may still be subject to various environment and adaptation risks at runtime. In this paper, we highlight the necessity and challenges of managing these risks. We overview existing work and present our efforts in identifying and controlling the risks. We argue that by managing these risks, the Internetware paradigm proceeds in a quality-assured direction.

1 Introduction

Nowadays more and more software systems are situated in open and dynamic environments such as the Internet or some cyber-physical scenarios. These systems are often characterized with autonomy, coordination, context-awareness and self-adaptability [1,2]. They pose new challenges for software developers and researchers, as conventional software engineering paradigms, including structured methods and Object-Oriented methods, fall short in managing the complexity of dealing with these new characteristics.

To this end, a new software paradigm, named *Internetware*, has been conceptualized and developed by a group of Chinese researchers including us in the past decade [1,2,3]. Behind the various new architecture models, development methods, enabling techniques and operating platforms of the Internetware paradigm, there is an old principle of Separation-of-Concerns, *i.e.*, the separation of coordination logic from autonomous computing entities, adaptation rules from business behavior and environment perception from system function.

* This research is supported by the China 973 Program (No. 2009CB320702), the China 863 Program (No. 2013AA01A213) and the National Natural Science Foundation of China (No. 61272047, 61100038, 61021062). Chang Xu is also supported by the Program for New Century Excellent Talents in University, China (NCET-10-0486).

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 271–284, 2013.

While the Internetware paradigm does help in managing the complexity, there are still important risks to be understood and controlled during the construction and execution of these adaptive systems situated in open environments. Some of these risks are caused by the inherent unpredictability of the environments and the limitation of environment sensing mechanisms. Misunderstanding of environment changes could cause faulty system reactions. Other risks arise from the aggressive separations of concerns. Over-simplified view of interactions among the environment sensing, system adaptation and business logic would lead to non-optimal or even erroneous behavior.

In this paper, we try to identify and eliminate some of the most significant risks. Especially, we will discuss the temporal disorder risk and the context inconsistency risk in the perception of environmental changes, as well as the faulty adaptation risk and the dynamic update risk in the adaptation to the changes. Although we have explored some of the concrete techniques and algorithms elsewhere [4,5,6,7,8,9], this paper is our first attempt on the integral study of risk management for the Internetware paradigm.

In the rest of this paper, we first introduce the Internetware paradigm and identify related risks in Section 2, then overview existing work and present our efforts in Section 3, and conclude the paper with a brief summary in Section 4.

2 Environment-Driven Internetware Model and Its Risks

2.1 Environment-Driven Architecture for Internetware

While complexity is in the essence of software [10,11], software application systems situated in open environments are more complex than traditional ones because: 1) they must dynamically discover and coordinate with autonomous external systems and resources, and 2) they have to deal with at runtime constant changes in their environment and in the requirements they must satisfy.

Developing such a system with conventional software engineering methods would suffer from this complexity because those methods normally assume, explicitly or implicitly, that during the execution of a system its environment and requirements remain stable and that the system can control, or at least safely assume about, the behavior of all resources it uses. If there were any dynamic environment/requirement changes to be handled at runtime, the system's developer should have fully predicted the changes, taken them as part of system inputs and hard-coded the change handling logic together with functional logic. Unfortunately, due to the ever changing nature of open environments, this approach would make the system over-complicated, even infeasible at all.

We need a new software paradigm to address this complexity directly. The basic idea of our approach to Internetware so far is two-fold. First, we separate coordination logic from business logic and reify it as a runtime model object, so that the coordination logic becomes programmable, and is also dynamically adaptable to the autonomous coordination entities. Second, we make the sensing of and adaptation to the environmental changes an explicit and integral part of the system, so that these concerns can be separately addressed while

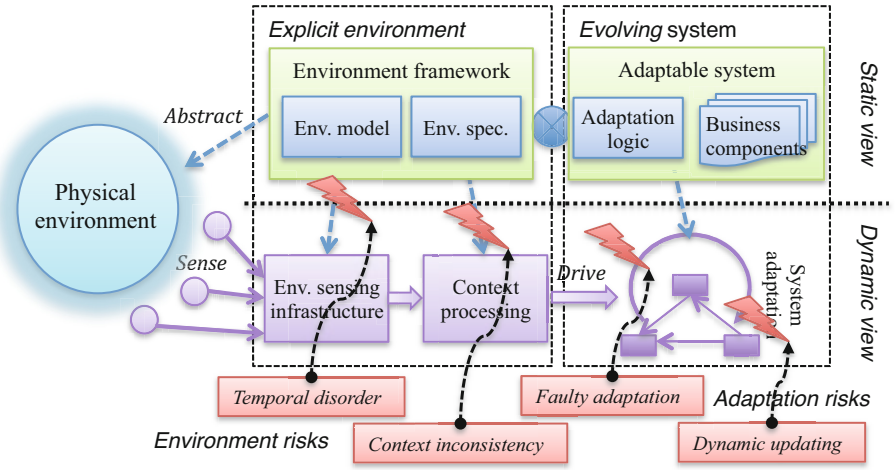


Fig. 1. Risks in the environment-driven Internetware architecture

their impacts on system functionalities can be systematically managed. These two treatments mutually premise each other. By the former the system becomes loosely-coupled and dynamically adjustable, which makes the adaptation to environment changes possible. By the latter the system as a whole can be viewed as an autonomous entity subject to further coordination.

For simplicity, in this paper we focus on the perception of environment changes and the dynamic system adaptation to these changes, as we have extensively discussed the architecture models and enabling techniques for the realization of the idea [12,13,2,14]. Fig. 1, which is adapted from [14], shows an environment-driven view of the Internetware architecture. In this view, an Internetware application system consists of two major parts:

Explicit Environment Constructs. This part is responsible for the modeling, sensing and understanding of environmental context, and reports interested environment change events to the evolving system part. Statically there is a domain-specific environment model that defines the scope and the laws of the environment to be sensed, and an application-specific environment specification that defines the contexts/events of interest to the current application. At runtime the concrete information about the environment is sensed and processed according to the environment model and specification, normally using some middleware infrastructure.

Dynamically Evolving System. This part provides the business functionality of the application. Different from conventional systems, it must be able to work with different configurations corresponding to different environment settings, and the switching between these configurations must be carried out smoothly at runtime. At runtime the system reacts to the reported environment changes by dynamically changing its structure and behavior as specified by the adaptation logic.

Here, the adaptation logic can be a set of rules or strategies [15,16] that decide whether or not the current configuration is still feasible, as well as if not feasible, how to evolve from one configuration to another. As this part is flexible and customizable by developers, it may be subject to risks as we explain later.

2.2 Identifying Risks

Conceptually, the architecture in Fig. 1 covers all major building blocks and operation steps of Internetware systems. However, things may not work exactly as expected when we zoom into the details. To prevent the system from deviating from its goal and to assure user satisfaction, some new risks beyond the consideration of conventional software engineering methods must be carefully handled. Among them there are:

Environment Risks. This category of risks can happen during the environment sensing and context processing steps. They come from the nature of dynamic and unpredictable physical environment, the noise and inaccuracy of sensing mechanisms used and the incorrect interpretation of sensed data. For example,

- *Temporal disorder risk.* Open computing environments are distributed in nature. To obtain a correct understanding of the environment from a collection of locally observed and partially ordered events cannot succeed unless these events are sufficiently ordered. This is non-trivial because global time is not always available in open environments.
- *Context inconsistency risk.* During the sensing step, environment data (a.k.a. contexts) are often sensed with different devices, on different time and at different locations. They also go through processing steps with possibly different algorithms. They are thus subject to various inconsistencies when they arrive at applications.

Adaptation Risks. This category of risks can happen during the adaptation decision and execution steps. Dynamic system adaptations must guarantee some level of robustness, stability and system consistency, despite the fact that the dynamics of the environment is not fully predictable. Adaptation risks threaten these guarantees. For example,

- *Faulty adaptation risk.* This risk is caused by developers' inadequate consideration of environmental dynamics. Although one can always reduce the risk by considering more cases that could happen, one cannot fully eliminate it because not everything can be predicted. Some sort of "safe nets" must be provided to control this risk.
- *Dynamic update risk.* This risk happens when the system is actually modified at runtime. We have learned a lot about the difficulties of (offline) software evolution, but dynamic update poses more challenges such as how to migrate the legacy state and how to ensure the system consistency and the correctness of on-going activities, not to mention the disruption caused by the update.

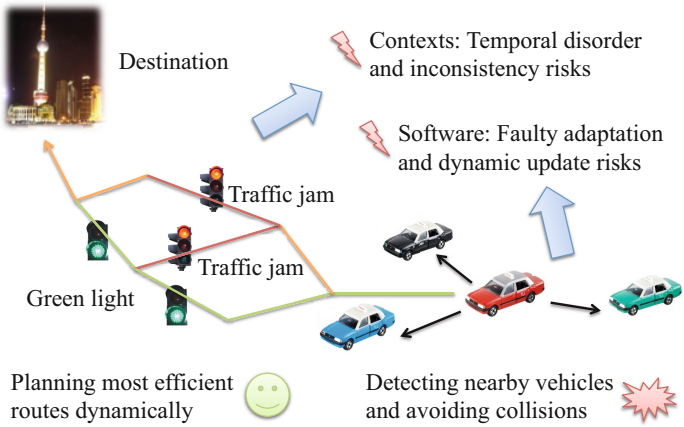


Fig. 2. A scenario of auto-piloting in smart city

Here we should mention that in this paper we use the term risk in a rather casual way. It means potential threats to software systems that may lead to failures or degradation of service quality. There are more formal treatments of risks in the software engineering community. For example, Cailliau and van Lamsweerde proposed a probabilistic framework for risk assessment in the requirement analysis phase [17]. More generally, Ishimatsu et al. advocated that the risk analysis process should be weaved into the whole lifecycle of system engineering in STPA [18]. However, we believe that the efforts presented below provide a first step to the full-fledged risk management for Internetware systems.

2.3 An Illustrating Scenario

Let us motivate our risk discussions with a scenario as illustrated in Fig. 2. Consider an auto-pilot application that controls a vehicle to travel in a smart city [19]. The vehicle is supported by a variety of sensing devices, either installed on the vehicle or deployed all over the city. The auto-pilot application aims to provide a safe and comfortable ride for passengers. It automatically avoids nearby vehicles to prevent collisions, and plans shortest routes without traffic jams to its destinations. All these can be done by collecting sensory data about the vehicle's surroundings, as well as measuring traffic conditions along its planned routes.

Software development for such kind of cyber-physical systems is a typical target domain of the Internetware paradigm because the constituting entities are autonomous, the environment is open and constantly changing, the system is built with the coordination of these autonomous entities and needs to automatically adapt to the changing environment.

For this scenario, the auto-pilot application can own the following adaptation logic. The vehicle takes its normal functionality of calculating a shortest route to its destination. If the traffic conditions on its calculated route deteriorate greatly as the vehicle travels, the application would adapt to new application

logic that recalculates a more efficient route based on actual traffic conditions. Later, during its journey, if some other vehicles on highways get too close to this vehicle, the application would again adapt to new adaptation logic that protects this vehicle by automatically avoiding potential collisions with nearby vehicles. As such, environmental changes would trigger the adaptation of this application in right time and keep guaranteeing its service quality.

However, the reality is not smooth. Sensing devices are distributed all over the city. They are not fully synchronized. The transmission of sensory data may also take unpredictable time before they arrive at a central server for processing. As such, collected sensory data are subject to varying delays, causing natural asynchrony. The asynchrony can easily incur temporal disorder of contexts derived from these sensory data. This can further mislead the vehicle to make wrong plans on travelling routes. This is one example of our discussed temporal disorder risk. Another context inconsistency risk may arise due to a similar concern. Contexts derived from sensory data are from different sensing devices. They work in an independent way, using different processing modules and adopting different local views to understand their environments. Their understanding can be incomplete, inaccurate, or even conflict with each other. Thus resulting context inconsistency would cause the vehicle to wrongly estimate its surrounding traffic conditions. Then preventing vehicle collisions cannot be fully or effectively ensured, when such context inconsistency risk is present.

More risks can arise when new adaptation logics are designed to cope with such environmental dynamics and risks. Adaptation logics can be faulty if exceptional cases are not adequately or properly considered. The faults are hard to detect by traditional model checking or testing approaches. Thus, adaptation allows an application to address environmental dynamics in a systematic way, but at the same time also makes the application error-prone. For the auto-pilot application, its route planning may work individually, but can be affected or even ruined by its adaptive collision avoidance strategy. Even if the application works for one vehicle, it may become unpredictable in effectiveness when deployed to thousands of vehicles. This is one example of our discussed faulty adaptation risk. Another dynamic update risk closely relates to the same concern. If the auto-pilot application is found to be faulty, it is impractical to reclaim all vehicles and reinstall a new version of this application manually. A better way is to automatically download the new version, and dynamically upgrade to it. Thus, dynamic update risk may arise, and guaranteeing safety for the update process when the vehicle is running is challenging.

3 Managing Risks

3.1 Temporal Disorder Risk

The contexts are distributed by nature. They are often collected by distributed but coordinating devices. Observe that context collecting devices may not have global clocks and may run at different speeds. They heavily rely on wireless communications, which suffer from finite but arbitrary delay. Moreover, context

collecting devices may postpone the dissemination of context data due to resource constraints, which also results in asynchrony. Some existing schemes for context processing implicitly assume that context collecting devices share the same notion of time [20,9]. This assumption does not necessarily hold in open environments. In some schemes, time is considered a very important type of context. In order to accurately compare and combine context arriving from distributed context collecting devices, these devices must share the same notion of time and be synchronized to the greatest extent possible [21]. In these schemes, the context collecting devices are simply assumed to have the same notion of time, or synchronization is conducted in a best-effort manner. This makes the environment-driven Internetware systems prone to the risk of perceiving temporally disordered contexts.

In order to control the temporal disorder risk resulting from the intrinsic asynchrony among context collecting devices, two complementary approaches can be be adopted. We can conduct synchronization among context collecting devices, and explicitly model and control the inaccuracy of time synchronization. Meanwhile, in case that only temporal order among contextual events is sufficient, we can also use logical time to explicitly record and maintain the temporal happen-before relation among contextual events.

For example, clock synchronization in TrueTime [22] is implemented by a set of time master machines per data center and a time slave daemon per machine. All masters' time references are regularly compared against each other. Every daemon polls a variety of masters to reduce vulnerability to errors from any one master. Between synchronizations, both masters and slaves advertise a slowly increasing time uncertainty that is derived from conservatively applied worst-case clock drift.

In [4,5], the message exchange among context collecting devices is used to establish the happen-before relation among contextual events. Logical vector clocks are employed to encode and decode this happen-before relation. Based on the happen-before relation, the applications' concerns on the temporal patterns of contextual events occurrences can be delineated by logic predicates. Online detection of such predicates enables the applications to be aware of the temporal properties of the contexts, despite the asynchrony.

Take the intersection management in the auto-pilot application as an example. Vehicles heading the same direction should pass the intersection one by one, while vehicles from directions cannot pass the intersection at the same time. Here each vehicle must sense the contextual events around it. More importantly, each vehicle must be guaranteed correct perception of the temporal order among contextual events. Or, the risk of temporal disorder of contexts may lead to traffic accidents.

3.2 Context Inconsistency Risk

The second environment risk concerns the inconsistency of contexts Internetware applications collect from environments. These contexts are firstly perceived by different sensing devices, and then composed by different software modules.

As mentioned earlier, these sensing devices have different local views of their environments, and software modules also have different built-in processing modules. As a result, the contexts finally collected by applications are subject to various inconsistencies. They would, if left unattended, drive these applications to behave abnormally.

Let us first consider the situation where context inconsistencies are not detected. Applications would use contexts from environments directly. They would use as many contexts as possible, aiming to understand their environments better. However, these contexts have inherent inconsistency problems, behaving as being incomplete, inaccurate, or even conflicting with each other [23,9]. This would affect application functionalities unexpectedly. Many existing context-aware applications [24,25] and middleware infrastructures [26,27,28,29] suffer from this problem.

To detect this risk, applications need to check collected contexts against consistency constraints [30,31,9], and use them only if no problems are found. Traditional inconsistency detection techniques [32,33,34,30] may apply here, but there is no guarantee that all contexts would be checked, or could be checked in time. This is because contexts are much more dynamic than traditional software artifacts like UML diagrams or XML documents. Without an efficient and effective methodology, contexts cannot be thoroughly checked for potential inconsistency problems at runtime. Therefore, we propose incremental or partial constraint checking techniques [35,9] for this purpose. They address two necessary requirements and corresponding challenges: 1) *efficient*, such that applications can have good responsiveness to dynamic contexts, and 2) *effective*, such that context inconsistency would not be missed and could be detected in time. By doing so, applications run with the enhanced ability of detecting context inconsistency risk at runtime.

A follow-up issue is that when the inconsistency risk is detected, it should be handled. This corresponds to resolution activities for detected context inconsistency. We note that these activities should be performed automatically and timely due to the massive and dynamic nature of contexts.

Let us also first consider the situation where context inconsistency risk is not handled. Contexts would be used by applications directly, or even if checked for inconsistency problems, the problems are simply recorded without any resolution. As a result, applications would run with inconsistent contexts, subject to various failures. For example, location-aware applications are very popular nowadays [36,37,38], but they are commonly suffering from the context inconsistency risk.

To handle this risk, context inconsistency should be resolved automatically and timely, such that resulting contexts fed to applications are guaranteed to be inconsistency-free. However, since resolution activities alter contexts, they inevitably affect application behaviors. Thus incurred side effect [39,40] may not have been considered or controlled. As a result, applications can still behave unexpectedly due to such uncontrolled resolution activities, leading to new challenges to protecting application functionalities. Therefore, we propose to resolve

context inconsistency, and at the same time measure and control the side effect of doing so. Then application-specific needs on context use are modeled and customized in resolution activities, such that a balance between inconsistency resolution and its incurred side effect can be achieved and optimized for each application [39,40]. This practice imposes new requirements and corresponding challenges in efficiently estimating and deciding potential side effect for different resolution activities at runtime. By doing so, applications run with the enhanced ability of handling inconsistency risk at runtime. Together with the earlier inconsistency detection ability, we are able to control context inconsistency risk for Internetware applications in a defensive way.

3.3 Faulty Adaptation Risk

The second adaptation risk concerns potential faults that may be experienced by Internetware applications during their runtime adaptation. Even with contexts free of inconsistency (i.e., inconsistency risk is controlled), applications can still be subject to various runtime failures, which only manifest in context-aware adaptation. These failures are due to their responsible faults in applications, which are caused by developers' inadequate consideration of environmental dynamics. Studies [16,41,8] show that these faults (named adaptation faults) differ from those that can be easily exposed by traditional software testing and analysis techniques. They would, if left unattended, drive applications to behave in an unpredictable or unstable way.

Let us first consider the situation where the faulty adaptation risk is not controlled. Applications are examined by traditional software testing and analysis techniques for simulated environments. Detected faults are then fixed. After that, these applications are released to real environments, where new adaptation faults can be encountered at runtime. This is because real environments are subject to various sensing noises, which cannot be completely predicted or precisely modeled in advance.

To protect application functionalities and prevent applications from suffering unexpected failures at runtime, the faulty adaptation risk needs to be controlled. Efforts can be made to exhaustively search an application's state space. Then all adaptation faults can be detected as long as they fall in this space [16]. However, such a state space is typically large or even infinite. Then the space is usually incompletely modeled or searched. As such, false negatives (i.e., missing adaptation faults) naturally result. On the other hand, since environmental dynamics is not fully predictable, application modeling can easily overlook or violate physical laws or domain rules. Then the modeling would fail to isolate infeasible scenarios and mistakenly include them into the fault analysis. This leads to numerous false positives (i.e., unreal adaptation faults) [42,8].

Based on these existing efforts, we propose to strengthen both dynamic analysis and static analysis to better detect adaptation faults. Dynamic analysis can incorporate runtime information, such that false negatives can be avoided [42,8]. Static analysis can prune infeasible scenarios by learned environment models, such that false positives can be alleviated [15]. Both analysis techniques contribute

to controlled adaptation risk for Internetware applications. Furthermore, even if missed adaptation faults manifest at runtime, they can also be restricted on their impact by automated application recovery [43]. This further strengthens an application's ability to control its experienced faulty adaptation risk.

3.4 Dynamic Update Risk

Once it is decided that runtime adaption is necessary to deal with detected changes in the environment, two issues arise. First, we need to decide the new configuration (or version) of the system that is suitable for the changed environment. Second, we need to figure out how to transfer the running system from the current configuration to the new one without shutting down the system or sacrificing ongoing activities. While the former can be addressed more or less by conventional techniques, the latter is more challenging and prone to risks.

The most severe risk here is that the correctness of the running system could be damaged if the dynamic update were not well coordinated with ongoing activities, even though the new configuration were guaranteed to be correct by itself [44,45]. For a simple instance, one may want to upgrade an encryption/decryption module with a new algorithm to prevent a security vulnerability just revealed. Dynamically replacing the old module with the new version could leave some of the tokens encrypted with the old algorithm to be decrypted by the new algorithm. This kind of problems would not happen at all if the update were carried out offline.

The second risk is that a naive solution to the above problem, which simply blocks all parts of the system that might be affected during the update, would introduce significant disruption to the service of the running system and diminish the benefits of dynamic update.

To strike a reasonable balance between system consistency and update disruption without introducing significant complexity for developers/administrators is far from trivial. Quiescence [44] and Tranquility [46] are two major existing proposals. Unfortunately, with the quiescence approach the disruption of dynamic update could still be too high; and with the tranquility approach global consistency of the system could still be compromised.

We proposed a new approach, named Version-Consistency, which ensures that distributed transactions be served as if they were operating either entirely on the old version or entirely on the new version, despite possible dynamic update that may happen meanwhile [6]. This approach provides better timeliness and lower disruption of dynamic update than Quiescence without sacrificing global system consistency. In addition to this component-level approach, we also developed a code-level dynamic update supporting system named Javelus [7]. It can support the dynamic update of Java programs running on industry-strength JVMs.

3.5 Towards an Integral Solution

To support the development, deployment, execution and adaptation of Internetware applications, and especially to help manage the environment and adaptation

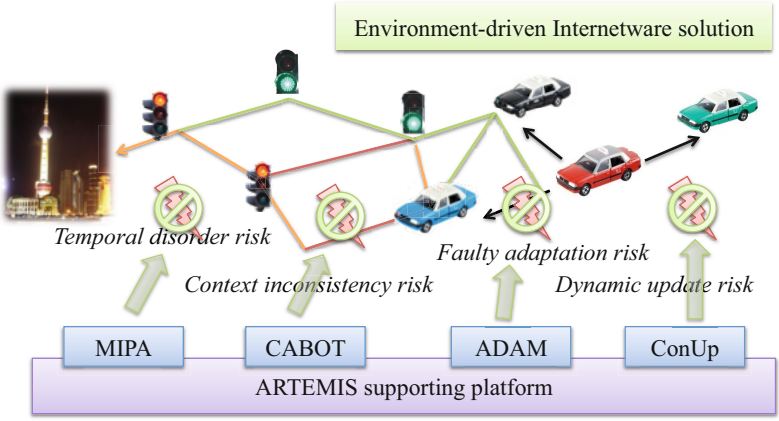


Fig. 3. Auto-piloting scenario supported by Artemis platform

risks, we have developed a set of tools and systems collectively branded with the name ARTEMIS. Among them, corresponding to the aforementioned four kinds of risks, there are

- MIPA the tool for distributed predicate detection,
- CABOT the middleware for context inconsistency detection and resolution,
- ADAM the framework re-synchronizing faultily adapted system to the current environment, and
- ConUp the support system for consistent and low-disruptive dynamic update.

Returning to the scenario discussed in §2.3, one can thus manage its contained risks with the support of our ARTEMIS platform. For example, MIPA evaluates global contextual predicates over partially ordered contextual events, which eliminates the risks of faulty interpretation of surrounding traffic conditions due to temporal disorder. CABOT deploys an efficient partial constraint checking engine to detect inconsistencies from location, obstacle and traffic condition contexts and resolve them automatically. ADAM identifies collision avoidance faults in context-aware adaptation, and recovers the vehicle from runtime adaptation errors. ConUp realizes dynamic upgrading of software on the vehicles in a seamless and safe way.

4 Summary

In this paper, we discussed the management of environment and adaptation risks for the Internetware paradigm. In particular, we studied the identification and controlling of temporal disorder and context inconsistency risks in the perception of environment changes, as well as faulty adaptation and dynamic update risks in the adaptation to perceived environment changes. We showed that managing these risks is important and necessary for guaranteeing functional and non-functional qualities for Internetware applications.

A solid software engineering paradigm must be built upon not only practical technology but also formal theory. In our work, we have exploited some formal models and mathematical tools to support the management of environment and adaptation risks. However they are used in a somewhat *ad hoc* manner. We plan to investigate some unified formal theory to help manage the risks and build high quality Internetware systems, following the style of the seminal work by Professor Jifeng He [47,48].

Acknowledgements. The authors would like to thank Xianping Tao, Chun Cao, Hao Hu and Ping Yu for their contributions to the development of the Internetware paradigm at the Institute of Computer Software, Nanjing University. We also like to thank the anonymous reviewer for his constructive comments and suggestions.

References

1. Yang, F., Lü, J., Mei, H.: Some discussion on the development of software technology. Acta Eletronica Sinica 26(9), 1104–1115 (2003)
2. Lü, J., Ma, X., Huang, Y., Cao, C., Xu, F.: Internetware: a shift of software paradigm. In: Proceedings of the First Asia-Pacific Symposium on Internetware, Internetware 2009, pp. 7:1–7:9. ACM, New York (2009)
3. Mei, H., Huang, G., Xie, T.: Internetware: A software paradigm for internet computing. Computer 45(6), 26–31 (2012)
4. Huang, Y., Yang, Y., Cao, J., Ma, X., Tao, X., Lu, J.: Runtime detection of the concurrency property in asynchronous pervasive computing environments. IEEE Transactions on Parallel and Distributed Systems 23(4), 744–750 (2012)
5. Yang, Y., Huang, Y., Cao, J., Ma, X., Lu, J.: Formal specification and runtime detection of dynamic properties in asynchronous pervasive computing environments. IEEE Trans. Parallel Distrib. Syst. (accepted in August 2012)
6. Ma, X., Baresi, L., Ghezzi, C., Panzica La Manna, V., Lu, J.: Version-consistent dynamic reconfiguration of component-based distributed systems. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE 2011, pp. 245–255. ACM, New York (2011)
7. Gu, T., Cao, C., Xu, C., Ma, X., Zhang, L., Lu, J.: Javelus: A low disruptive approach to dynamic software update. In: Proceedings of the 19th Asia-Pacific Software Engineering Conference, APSEC 2012, pp. 527–536 (2012)
8. Xu, C., Cheung, S., Ma, X., Cao, C., Lu, J.: Adam: Identifying defects in context-aware adaptation. Journal of Systems and Software 85(12), 2812–2828 (2012)
9. Xu, C., Cheung, S.C., Chan, W.K., Ye, C.: Partial constraint checking for context consistency in pervasive computing. ACM Trans. Softw. Eng. Methodol. 19(3), 1–61 (2010)
10. Brooks Jr., F.P.: No silver bullet essence and accidents of software engineering. Computer 20(4), 10–19 (1987)
11. Booch, G.: Object Oriented Analysis & Design with Application. Pearson Education India (2006)
12. Lü, J., Ma, X., Tao, X., Cao, C., Huang, Y., Yu, P.: On environment-driven software model for Internetware. Science in China Series F: Information Sciences 51(6), 683–721 (2008)

13. Ma, X., Cheung, S., Cao, C., Xu, F., Lu, J.: Towards a dependable software paradigm for service-oriented computing. In: Zhang, L.J., Paul, R., Dong, J. (eds.) High Assurance Services Computing, pp. 163–192. Springer US (2009)
14. Lü, J., Ma, X., Tao, X., Huang, Y., Xu, C.: Explicit environmental constructs for Internetware. Science Sinica Informationis 43(1), 1–23 (2013) (in Chinese)
15. Liu, Y., Xu, C., Cheung, S.: Afchecker: Effective model checking for context-aware adaptive applications. Journal of Systems and Software 86(3), 854–867 (2013)
16. Sama, M., Elbaum, S., Raimondi, F., Rosenblum, D.S., Wang, Z.: Context-aware adaptive applications: Fault patterns and their automated identification. IEEE Transactions on Software Engineering 36, 644–661 (2010)
17. Cailliau, A., van Lamsweerde, A.: A probabilistic framework for goal-oriented risk analysis. In: Proceedings of the 2012 IEEE 20th International Requirements Engineering Conference, RE 2012, pp. 201–210. IEEE Computer Society, Washington, DC (2012)
18. Ishimatsu, T., Leveson, N., Thomas, J., et al.: Modeling and hazard analysis using stpa. In: Proceedings of the Conference of the International Association for the Advancement of Space Safety, pp. 1–10 (2010)
19. Guizzo, E.: How Google's self-driving car works, `http://spectrum.ieee.org/automaton/robotics/artificial-intelligence/how-google-self-driving-car-works` (last accessed in March 2013)
20. Huang, Y., Ma, X., Tao, X., Cao, J., Lu, J.: A probabilistic approach to consistency checking for pervasive context. In: Proc. IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, EUC 2008, Shanghai, China, pp. 387–393 (December 2008)
21. Dey, A.: Providing architectural support for building context-aware applications. PhD thesis, Georgia Institute of Technology (2000)
22. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally-distributed database. In: Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation, OSDI 2012, pp. 251–264. USENIX Association, Berkeley (2012)
23. Xu, C., Cheung, S.C.: Inconsistency detection and resolution for context-aware middleware support. In: Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2005, Lisbon, Portugal, pp. 336–345 (September 2005)
24. Malan, D., Fulford-Jones, T., Welsh, M., Moulton, S.: Codeblue: An ad hoc sensor network infrastructure for emergency medical care. In: Proc. Mobisys Workshop on Applications of Mobile Embedded Systems, Boston, MA, USA, pp. 12–14 (June 2004)
25. Ranganathan, A., Campbell, R., Ravi, A., Mahajan, A.: Conchat: a context-aware chat program. IEEE Pervasive Computing 1(3), 51–57 (2002)
26. Capra, L., Emmerich, W., Mascolo, C.: Carisma: Context-aware reflective middleware system for mobile applications. IEEE Transactions on Software Engineering 29(10), 929–945 (2003)
27. Julien, C., Roman, G.C.: Egospaces: Facilitating rapid development of context-aware mobile applications. IEEE Transactions on Software Engineering 32(5), 281–298 (2006)
28. Murphy, A.L., Picco, G.P., Roman, G.C.: Lime: A coordination model and middleware supporting mobility of hosts and agents. ACM Trans. on Software Engineering and Methodology 15(3), 279–328 (2006)

29. Ranganathan, A., Campbell, R.H.: An infrastructure for context-awareness based on first order logic. Personal Ubiquitous Comput. 7, 353–364 (2003)
30. Reiss, S.: Incremental maintenance of software artifacts. IEEE Transactions on Software Engineering 32(9), 682–697 (2006)
31. Tarr, P., Clarke, L.: Consistency management for complex applications. In: Proceedings of the 20th International Conference on Software Engineering, pp. 230–239 (1998)
32. ARGOUML, `http://argouml.tigris.org/` (last accessed in March 2013)
33. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: ACM/IEEE 30th International Conference on Software Engineering, ICSE 2008, pp. 511–520 (2008)
34. Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A.: xlinkit: a consistency checking and smart link generation service. ACM Trans. on Internet Technology 2(2), 151–185 (2002)
35. Xu, C., Cheung, S.C., Chan, W.K.: Incremental consistency checking for pervasive context. In: Proc. International Conference on Software Engineering, ICSE 2006, Shanghai, China, pp. 292–301 (May 2006)
36. Locale, `http://www.twofortyfouram.com/` (last accessed in March 2013)
37. Setting Profiles, `http://www.probeez.com/` (last accessed in March 2013)
38. Tasker, `http://tasker.dinglisch.net/` (last accessed in March 2013)
39. Xu, C., Cheung, S.C., Chan, W.K., Ye, C.: On impact-oriented automatic resolution of pervasive context inconsistency. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE 2007, pp. 569–572. ACM, New York (2007)
40. Xu, C., Ma, X., Cao, C., Lu, J.: Minimizing the side effect of context inconsistency resolution for ubiquitous computing. In: Puiatti, A., Gu, T. (eds.) MobiQuitous 2011. LNICST, vol. 104, pp. 285–297. Springer, Heidelberg (2012)
41. Sama, M., Rosenblum, D.S., Wang, Z., Elbaum, S.: Multi-layer faults in the architectures of mobile, context-aware adaptive applications. Journal of Systems and Software 83(6), 906–914 (2010)
42. Xu, C., Cheung, S.C., Ma, X., Cao, C., Lu, J.: Dynamic fault detection in context-aware adaptation. In: Proceedings of the Fourth Asia-Pacific Symposium on Internetware, Internetware 2012, pp. 1:1–1:10. ACM (2012)
43. Zhang, L., Xu, C., Ma, X., Gu, T., Hong, X., Cao, C., Lu, J.: Resynchronizing model-based self-adaptive systems with environments. In: Proceedings of the 19th Asia-Pacific Software Engineering Conference, APSEC 2012, pp. 184–193 (December 2012)
44. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Transactions on Software Engineering 16(11), 1293–1306 (1990)
45. Gupta, D., Jalote, P., Barua, G.: A formal framework for on-line software version change. IEEE Transactions on Software Engineering 22(2), 120–131 (1996)
46. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. IEEE Transactions on Software Engineering 33(12), 856–868 (2007)
47. He, J., 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)
48. Hoare, C.A.R., He, J.: Unifying theories of programming. Prentice Hall (1998)

Safety versus Security in the Quality Calculus

Hanne Riis Nielson and Flemming Nielson

DTU Compute, Technical University of Denmark, Denmark
{hrni,fnie}@dtu.dk

Abstract. Safety and security are both needed for ensuring that cyber-physical systems live up to expectations, but often an intelligent trade-off is called for, because sometimes it is impossible to obtain optimal safety at the same time as optimal security. In the context of the Quality Calculus we develop a *type system* for checking the extent to which safety and security goals have been met.

Safety goals include showing that certain error configurations are in fact not reachable and hence do not require intelligent error handling.

Security goals include showing that highly trusted communications can only be performed in highly trusted contexts. This is potentially too demanding and the Quality Calculus is therefore extended with a primitive for endorsing data to a higher trust level (accepting violations of the explicit flow) and for temporarily asserting a higher trust in the context (accepting violations of the implicit flow).

This is illustrated on a *worked example* taken from the automotive sector and we conclude with a discussion of the *theoretical properties* of the type system.

1 Introduction

Motivation. One of the challenges of cyber-physical systems [8] is to reconcile the often conflicting demands of security and safety. *Safety* concerns making sure that no harm can arise from using the systems; as an example an airplane should continue flying, and a car should continue to react to braking and steering. *Security* concerns making sure that nobody can pretend to be somebody else; as an example, steering directions should originate from the steering wheel and not from a car game played by the children on the back seat. These demands often conflict because safety requires very fast reaction to alerts whereas security requires that sufficient time is taken to ensure the authenticity (or integrity) of the alert. Yet both demands need to be addressed in order for the overall cyber-physical system to live up to expectations and hence intelligent trade-offs are called for.

The design of cyber-physical systems grows out of the design of embedded systems and is often performed by companies having a strong safety culture. They are often well trained in using general safety standards as well as more specialised safety standards for automotive, aeronautic, or health care applications. As communications increasingly become wireless or are multiplexed over optical fibres or electrical wires, there is a clearly identified need to incorporate

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 285–303, 2013.

security in order to prevent cyber attacks on systems. While it is recognized that both safety and security are important, one often hears the slogan that in the time of crisis *safety takes precedence over security*.

We believe that this slogan reflects an unclear understanding of the many-faceted nature of security. We might agree that in the time of crisis *safety takes precedence over confidentiality*; it would be imprudent not to communicate warnings about safety problems merely because these warnings can be overheard by others. However, to challenge the slogan of the safety community, we would like to suggest that *authenticity (and integrity) takes precedence over safety*; it would be imprudent not to check that orders to change course do in fact originate from authorised sources rather than (unaware or malicious) intruders.

Another maxim of the safety culture is that systems should be *fail-safe*; this means that systems either do not fail or that they fail in a particularly graceful manner having as few safety implications as possible. Taking the unreliability of communication into account this means that entire systems are designed around instructions of the form "please feel free to continue reversing (at the current or lower speed) for the next 3 seconds" rather than "please feel free to continue reversing (at the current or lower speed) until instructed otherwise", because if communication breaks down the former would lead to much fewer safety incidents than the latter; the design of the European Railway Train Management System (ERTMS) is a case in point.

This directly relates to the risk of *denial of service* as considered in security. While attacks on the confidentiality, integrity and authenticity of messages can be averted through the proper use of cryptographic communication protocols, there is hardly any feasible way to guarantee against denial of service attacks in cyber-physical systems. The reason is that cyber-physical systems are by their very nature open and hence wireless communication can be jammed and optical fibers and electrical cables can be cut. This suggests that the proper way to deal with denial of service attacks is to ensure that systems are developed in such a way that the consequences of communication break-down are as benign as possible.

Contribution. In a previous paper [12] we proposed a process calculus, the Quality Calculus, that allows to express due care in always having default or substitute data available in case the real data cannot be obtained due to unreliable communication. The development was facilitated by a SAT-based [9,4] robustness analysis to determine whether or not undesirable error configurations could in fact be avoided by always choosing alternative configurations possibly using default or substitute data. This addresses the issue of how best to secure systems against *denial of service* and to obtain overall *fail-safe* behaviour.

In this paper we extend the Quality Calculus with trust levels indicating the degree of trust we can have in the authenticity and integrity of communications. In the interest of simplicity (and in accordance with our slogan that *authenticity takes precedence over safety*) we shall ignore all issues relating to confidentiality of communications. Instead we shall develop analyses to identify the extent to which a system is *robust* against the following two types of attacks:

- *denial of service attacks* due to other processes experiencing faults of their own or due to an attacker disrupting the communication, for example by jamming, and
- *integrity attacks* due to an attacker breaking part of the cryptographic communication protocol, for example by a brute force attack on a weak cryptosystem or through the recovery of current session keys by physically dismantling and inspecting devices in the cyber-physical system.

Our aim is to identify those points in systems where decisions are made to increase the integrity level, either through

(i) the endorsement of data (so as to allow an explicit flow of information otherwise prohibited), or through
(ii) the temporary assertion of a higher trust-level for the context (the"program counter" so as to allow an implicit flow of information otherwise prohibited),

and to identify those points where the proper use of default or substitute data succeeds in

(iii) establishing the non-reachability of certain "error" configurations.

We do so by introducing language primitives to indicate the three points of consideration and a safety and security type system enforcing that points of type *(i)* and *(ii)* have not been forgotten and that points of type *(iii)* have not be introduced without due care. The safety and security system amalgamates an adaptation of the SAT-based robustness analysis of [12] with type systems for integrity, which are dual to type systems for confidentiality (e.g. [16]).

Overview. Section 2 defines the syntax of the Quality Calculus, Section 3 defines the semantics in both closed and open environments, Section 4 defines the safety and security type system, Section 5 contains a worked example, Section 6 discusses the theoretical properties of the system, and finally Section 7 concludes.

2 Syntax of the Quality Calculus

Our starting point is the Quality Calculus introduced in [12]. It is a calculus in the π-calculus family [11] but extended with constructs allowing one to take appropriate measures in case of unreliable communications. It shares this aim with a number of other calculi studying how to model faults in the underlying communication network (as for example [1,2,5,6,14]) but is doing so at a somewhat higher abstraction level. The main novelty of the Quality Calculus is a binder construct simultaneously waiting for a number of inputs and using a quality predicate to determine when sufficient inputs have been received and thereby when to continue with the computations.

We now extend the Quality Calculus [12] with notation for both safety and security. We shall be based on a lattice $\mathcal{L}$ of *trust levels* indicating the level of integrity or authenticity of the communications; we shall sometimes use $\mathcal{L}^\top$ where $\mathcal{L}$ is extended with a new greatest element $\top$ and $\mathcal{L}^\top_\bot$ where $\mathcal{L}^\top$ is extended

Table 1. Terms t, expressions e, binders b, processes P, and configurations Q

$$
\begin{array}{rcl}
t & ::= & y \mid c \mid f(t_1, \cdots, t_n) \mid \mathsf{endorse}^{\ell_2}_{\ell_1}(t) \\
e & ::= & x \mid \mathsf{some}(t) \mid \mathsf{none} \\
b & ::= & t_1!^{\ell} t_2\{x\} \mid t_1?^{\ell} x\{t_2\} \mid \&_q(b_1, \cdots, b_n) \mid \mathsf{assert}^{\ell_2}_{\ell_1}(b) \\
P & ::= & (\nu c)\, P \mid P_1 \,|\, P_2 \mid \mathsf{0} \mid b.P \mid A(e) \\
 & & \mid \mathsf{case}^{\ell}\ e\ \mathsf{of}\ \mathsf{some}(y) \colon P_1\ \mathsf{else}\ P_2 \mid \mathsf{dummy}(P) \\
Q & ::= & (\nu c)\, Q \mid Q_1 \,|\, Q_2 \mid \mathsf{0} \mid \langle \ell, P \rangle
\end{array}
$$

with a new least element $\bot$. As a simple example, $\mathcal{L}$ might be $\{\text{H}, \text{L}\}$ with $\text{L} \sqsubseteq \text{H}$ indicating that H is trusted more than L; more complex examples can be designed using Mandatory Access Controls systems focusing on integrity levels (e.g. [7]).

A *system* S consists of a number of process definitions, a main configuration Q_* and a number of global constants c_i:

$$
\begin{array}{ll}
\mathsf{define} & A_1(x_1) \triangleq P_1 \\
 & \quad \vdots \\
 & A_n(x_n) \triangleq P_n \\
\mathsf{in} & Q_* \\
\mathsf{using} & c_1, \cdots, c_k
\end{array}
$$

Terms, expressions, binders, processes and configurations are defined in Table 1. *Terms* t are used to construct *data* elements and consists of variables y, constants c, function applications $f(t_1, \cdots, t_n)$, and the ability to increase the trust level of terms $\mathsf{endorse}^{\ell_2}_{\ell_1}(t)$ thereby accepting violations of the explicit flow of information; here $\ell_1, \ell_2 \in \mathcal{L}$. *Expressions* e are used to construct *optional data* and consists of variables x, optional data that is available $\mathsf{some}(t)$, and optional data that is not available none.

Binders b express potentially unreliable communication. Output $t_1!^{\ell} t_2\{x\}$ indicates that data t_2 is sent over channel t_1 with x being the acknowledgement of failure or success of the communication (none means failure and $\mathsf{some}(t)$ means success). Input $t_1?^{\ell} x\{t_2\}$ indicates that x is the result of receiving from the channel t_1 and t_2 denoting the data to be sent as acknowledgement in case communication succeeds; a communication will result in x being bound to optional data (none in case of failure of communication and $\mathsf{some}(t)$ for the successful communication of the data t). For both output and input the trust level ℓ indicates the intended level of trust in the communication. The quality predicate q used in $\&_q(b_1, \cdots, b_n)$ expresses the overall criterion for an aggregate communication to be successful. Example quality predicates include $\forall$ and $\exists$ requiring that all components, resp. at least one component, must be successful. Finally, the construct $\mathsf{assert}^{\ell_2}_{\ell_1}(b)$ is the analogue of $\mathsf{endorse}^{\ell_2}_{\ell_1}(t)$ for raising the trust level in case of binders and thereby accepting violations of the implicit flow of information.

Processes P allow to create new fresh constants $(\nu c)\,P$, the parallel combination of processes $P_1 \mid P_2$, the empty process $\mathsf{0}$, communication $b.P$, and calling defined processes $A(e)$. The construct $\mathsf{case}^{\ell}\ e\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2$ allows to inspect the availability of the optional data e, choosing P_1 with y denoting t in case e yields $\mathsf{some}(t)$, and choosing P_2 in case e yields none. Finally, the construct $\mathsf{dummy}(P)$ indicates that the error handling expressed by P is in fact not necessary (due to the careful use of default and substitute data in case the real data is not available) and hence choosing a simple-minded $P = \mathsf{0}$ will be adequate.

Configurations Q allow to annotate processes with the trust level where they are expected to operate; this will correspond to the trust level of the context or "program counter". The basic construct is that of $\langle \ell, P \rangle$ indicating that the process P is intended to operate at trust level ℓ. The main configuration Q_* typically takes the form $\langle \ell_*, P_* \rangle$ where ℓ_* is the highest trust level in the lattice $\mathcal{L}$ and P_* is the main process of interest. On top of this we can combine configurations using some of the operators available for processes.

The syntax generalises that of [12] in the consideration of trust levels, the use of notation $\mathsf{endorse}^{\ell_2}_{\ell_1}(\cdots)$, $\mathsf{assert}^{\ell_2}_{\ell_1}(\cdots)$ and $\mathsf{dummy}(\cdots)$, the extension of communication to include acknowledgements, and the ability to use quality predicates not only on inputs but also on outputs. The latter is exploited in the simple example below.

Example 1. Let $\mathcal{L}$ be the lattice $\{\mathrm{H}, \mathrm{L}\}$ with $\mathrm{L} \sqsubseteq \mathrm{H}$ and consider the system

$$
\begin{array}{ll}
\mathsf{define} & A_1(x) \triangleq (\nu d_1)\,\&_{\forall}(c!^{\mathrm{H}} d_1\{z_1\}, c'?^{\mathrm{L}} x_1\{\checkmark\}).A_1(x) \\
& A_2(x) \triangleq (\nu d_2)\,\&_{\exists}(c?^{\mathrm{H}} x_2\{\checkmark\}, c'!^{\mathrm{L}} d_2\{z_2\}).A_2(x) \\
\mathsf{in} & \langle \mathrm{H}, A_1(\mathsf{none}) \mid A_2(\mathsf{none}) \rangle \\
\mathsf{using} & c, c', \checkmark
\end{array}
$$

declaring two processes communicating over the channels c and c' and exchanging the constant $\checkmark$ as an acknowledgement. Both processes are ready to input and output over the two channels; the channel c is used for highly trusted communications while c' is used for communications at a lower trust level. The process A_1 requires that both actions are performed before recursing while A_2 only requires that one of them are successful before recursing. □

3 Semantics in Open and Closed Environments

Communication is intended to be point to point between one sender and one receiver. In a closed environment both the sender and receiver will be part of the given system whereas in an open environment only one needs to be part of the specified system. We mostly follow the classical approach and define the semantics by a structural congruence expressing when two configurations are congruent to one another and a transition relation describing when one configuration evolves into another. The process definitions remain fixed throughout.

The structural congruence $Q_1 \equiv Q_2$ is specified in Table 2; it enforces that configurations constitute a monoid with respect to parallel composition and the

Table 2. The structural congruence

$Q \equiv Q$	$Q_1 \equiv Q_2 \Rightarrow Q_2 \equiv Q_1$	$Q_1 \equiv Q_2 \wedge Q_2 \equiv Q_3 \Rightarrow Q_1 \equiv Q_3$
$Q \,\vert\, \mathsf{0} \equiv Q$	$Q_1 \,\vert\, Q_2 \equiv Q_2 \,\vert\, Q_1$	$Q_1 \,\vert\, (Q_2 \,\vert\, Q_3) \equiv (Q_1 \,\vert\, Q_2) \,\vert\, Q_3$
$(\nu c)\, Q \equiv Q$ if $c \notin \mathsf{fc}(Q)$	$(\nu c_1)\,(\nu c_2)\, Q \equiv (\nu c_2)\,(\nu c_1)\, Q$	$(\nu c)\,(Q_1 \mid Q_2) \equiv ((\nu c)\, Q_1) \mid Q_2$ if $c \notin \mathsf{fc}(Q_2)$
$Q_1 \equiv Q_2 \Rightarrow Q_1 \mid Q \equiv Q_2 \mid Q$	$Q_1 \equiv Q_2 \Rightarrow (\nu c)\, Q_1 \equiv (\nu c)\, Q_2$	
$\langle \ell, \mathsf{0} \rangle \equiv \mathsf{0}$	$\langle \ell, (\nu c)\, P \rangle \equiv (\nu c)\, \langle \ell, P \rangle$	$\langle \ell, P_1 \,\vert\, P_2 \rangle \equiv \langle \ell, P_1 \rangle \,\vert\, \langle \ell, P_2 \rangle$

Table 3. The transition relation for processes and systems

$$\frac{b_1 \xrightarrow{c_1!^{\ell} c_2\{c\}} b_1' \quad b_2 \xrightarrow{c_1?^{\ell} c_2\{c\}} b_2'}{\langle \ell_1, b_1.P_1 \rangle \mid \langle \ell_2, b_2.P_2 \rangle \xrightarrow{c_1\dagger^{\ell} c_2\{c\}} \langle \ell_1, P_1' \rangle \mid \langle \ell_2, P_2' \rangle} \text{ where } P_i' = \begin{cases} b_i'.P_i & \text{if } b_i' ::_{\mathsf{ff}} \theta_i \\ P_i\theta_i & \text{if } b_i' ::_{\mathsf{tt}} \theta_i \end{cases}$$

$$\frac{b \xrightarrow{c_1!^{\ell} c_2\{c\}} b'}{\langle \ell', b.P \rangle \xrightarrow{c_1!^{\ell} c_2\{c\}} \langle \ell', P' \rangle} \text{ where } P' = \begin{cases} b'.P & \text{if } b' ::_{\mathsf{ff}} \theta \\ P\theta & \text{if } b' ::_{\mathsf{tt}} \theta \end{cases}$$

$$\frac{b \xrightarrow{c_1?^{\ell} c_2\{c\}} b'}{\langle \ell', b.P \rangle \xrightarrow{c_1?^{\ell} c_2\{c\}} \langle \ell', P' \rangle} \text{ where } P' = \begin{cases} b'.P & \text{if } b' ::_{\mathsf{ff}} \theta \\ P\theta & \text{if } b' ::_{\mathsf{tt}} \theta \end{cases}$$

$$\frac{e \rhd \mathsf{some}(c)}{\langle \ell', \mathsf{case}^{\ell}\ e\ \mathsf{of\ some}(y)\colon P_1\ \mathsf{else}\ P_2 \rangle \xrightarrow{\tau} \langle \ell, P_1[c/y] \rangle}$$

$$\frac{e \rhd \mathsf{none}}{\langle \ell', \mathsf{case}^{\ell}\ e\ \mathsf{of\ some}(y)\colon P_1\ \mathsf{else}\ P_2 \rangle \xrightarrow{\tau} \langle \ell, P_2 \rangle}$$

$$\frac{e \rhd w}{\langle \ell, A(e) \rangle \xrightarrow{\tau} \langle \ell, P[w/x] \rangle} \text{ where } A(x) \triangleq P \text{ and } w = \mathsf{some}(c) \text{ or } w = \mathsf{none} \qquad \langle \ell, \mathsf{dummy}(P) \rangle \xrightarrow{\tau} P$$

$$\frac{Q_1 \equiv Q_2 \quad Q_2 \xrightarrow{\alpha} Q_3 \quad Q_3 \equiv Q_4}{Q_1 \xrightarrow{\alpha} Q_4} \qquad \frac{Q_1 \xrightarrow{\alpha} Q_2}{Q_1 \mid Q \xrightarrow{\alpha} Q_2 \mid Q}$$

$$\frac{Q \xrightarrow{\alpha} Q'}{\mathsf{define}\ \cdots\ \mathsf{in}\ (\nu \bar{c'})\, Q\ \mathsf{using}\ \bar{c} \xrightarrow{\alpha} \mathsf{define}\ \cdots\ \mathsf{in}\ Q'\ \mathsf{using}\ \bar{c}\bar{c'}\bar{c''}}$$

where $\{\bar{c'}\} \cap \{\bar{c}\} = \emptyset$ and $\mathsf{fn}(\alpha) \subseteq \{\bar{c}\bar{c'}\}$ and $\{\bar{c''}\} = \mathsf{bn}(\alpha) \setminus \{\bar{c}\bar{c'}\}$

empty process and it takes care of the scopes for constants; here $\mathsf{fc}(Q)$ is the set of constants occurring free in Q.

The transition relation $Q_1 \xrightarrow{\alpha} Q_2$ is defined in Table 3; here α will either be τ indicating that the transition does not involve any communication, or it

Table 4. The transition relation for binders

$$\frac{t_1 \rhd c_1 \quad t_2 \rhd c_2}{t_1!^{\ell}t_2\{x\} \xrightarrow{c_1!^{\ell}c_2\{c\}} [\mathsf{some}(c)/x : \ell]} \qquad \frac{t_1 \rhd c_1 \quad t \rhd c}{t_1?^{\ell}x\{t\} \xrightarrow{c_1?^{\ell}c_2\{c\}} [\mathsf{some}(c_2)/x : \ell]}$$

$$\frac{b_i \xrightarrow{\beta} b_i'}{\&_q(b_1, \cdots, b_i, \cdots, b_n) \xrightarrow{\beta} \&_q(b_1, \cdots, b_i', \cdots, b_n)} \qquad \frac{b \xrightarrow{\beta} b'}{\mathsf{assert}^{\ell_2}_{\ell_1}(b) \xrightarrow{\beta} \mathsf{assert}^{\ell_2}_{\ell_1}(b')}$$

$$t_1!^{\ell}t_2\{x\} ::_{\mathsf{ff}} [\mathsf{none}/x] \qquad t_1?^{\ell}x\{t_2\} ::_{\mathsf{ff}} [\mathsf{none}/x] \qquad [\mathsf{some}(c)/x : \ell] ::_{\mathsf{tt}} [\mathsf{some}(c)/x]$$

$$\frac{b_1 ::_{v_1} \theta_1 \quad \cdots \quad b_n ::_{v_n} \theta_n}{\&_q(b_1, \cdots, b_n) ::_v \theta_n \cdots \theta_1} \text{ where } v = [\![q]\!](v_1, \cdots, v_n) \qquad \frac{b ::_v \theta}{\mathsf{assert}^{\ell_2}_{\ell_1}(b) ::_v \theta}$$

will be $c_1\dagger^{\ell}c_2\{c\}$ indicating an internal communication of c_2 over the channel c_1 and with acknowledgement c, or it will be $c_1?^{\ell}c_2\{c\}$ or $c_1!^{\ell}c_2\{c\}$ indicating an external communication. The definition makes use of a relation $t \rhd c$ describing when a term t evaluates to a constant c and a similar relation describing when an expression e evaluates to a constant that either has the form $\mathsf{some}(c)$ or is none. The definitions of these relations are straightforward and will therefore be omitted; we shall only notice that the semantics is oblivious to the effect of the $\mathsf{endorse}$ construct as the trust levels are not part of the semantic values.

To handle the binders we make use of a transition relation $b \xrightarrow{\alpha} b'$ (defined in Table 4) recording that the binder b evolves into b' while performing the action α. Here α can be an output action $c_1!^{\ell}c_2\{c\}$ or an input action $c_1?^{\ell}c_2\{c\}$. The result b' will be a binder containing substitutions and we formalise this by extending the syntax of binders to

$$b ::= \cdots \mid [\mathsf{some}(c)/x : \ell]$$

where $[\mathsf{some}(c)/x : \ell]$ denotes that x is to be replaced by $\mathsf{some}(c)$; we retain the trust level of the communication for the purposes of establishing a subject reduction result (in Section 6) for our type system (defined in Section 4). Occasionally, we shall allow to write $b \xrightarrow{\alpha,x} b'$ to indicate that by inspection of the inference tree $b \xrightarrow{\alpha} b'$ one may observe that the step is due to the variable x being bound to a constant.

We also make use of the relation $b ::_v \theta$; it records (in $v \in \{\mathsf{tt}, \mathsf{ff}\}$) whether or not all required inputs of b have been performed and it records the corresponding substitution (θ). It is here the quality predicates q are used to check whether sufficient actions have been performed and we shall write $[\![q]\!]$ for the semantics of quality predicates; as an example we have $[\![\forall]\!](v_1, \cdots, v_n) = v_1 \wedge \cdots \wedge v_n$ and $[\![\exists]\!](v_1, \cdots, v_n) = v_1 \vee \cdots \vee v_n$. We shall write id for the identity substitution and $\theta_2\theta_1$ for the composition of two substitutions, so $(\theta_2\theta_1)(x) = \theta_2(\theta_1(x))$ for all x.

Finally, returning to Table 3, the transition relation $S_1 \xrightarrow{\alpha} S_2$ for systems is defined from the one for processes. It takes care of extending the list of known constants $\bar{c}$, with those that are *extruded* from the internals of the system $\bar{c'}$, as well as those that are extruded from the environment $\bar{c''}$. To express this we make use of $\mathsf{bn}(\alpha)$ to express those names in α that might be created by the environment; it is given by $\mathsf{bn}(\tau) = \emptyset$, $\mathsf{bn}(c_1 \dagger^\ell c_2\{c\}) = \emptyset$, $\mathsf{bn}(c_1!^\ell c_2\{c\}) = \{c\}$, and $\mathsf{bn}(c_1?^\ell c_2\{c\}) = \{c_2\}$. The renaming names of α are denoted $\mathsf{fn}(\alpha)$; they are given by $\mathsf{fn}(\tau) = \emptyset$, $\mathsf{fn}(c_1 \dagger^\ell c_2\{c\}) = \{c_1, c_2, c\}$, $\mathsf{fn}(c_1!^\ell c_2\{c\}) = \{c_1, c_2\}$, and $\mathsf{fn}(c_1?^\ell c_2\{c\}) = \{c_1, c\}$.

Example 2. Returning to Example 1 let us assume that the processes first engage in an internal highly trusted communication over the channel c; the action is $c \dagger^{\mathrm{H}} d_1\{\checkmark\}$ and the resulting configuration is:

$$\begin{array}{ll}
\mathsf{define} & A_1(x) \triangleq \cdots \\
& A_2(x) \triangleq \cdots \\
\mathsf{in} & \langle \mathrm{H}, \&_\forall([\mathsf{some}(\checkmark)/z_1 : \mathrm{H}], c'?^{\mathrm{L}} x_1\{\checkmark\}).A_1(\mathsf{none}) \\
& \quad \mid A_2(\mathsf{none})[\mathsf{some}(d_1)/x_1; \mathsf{none}/z_2]\rangle \\
\mathsf{using} & c, c', \checkmark, d_1, d_2
\end{array}$$

Thus the first process records the result of the first communication in its binder whereas the second process is ready to continue.

In the next step the system can interact with the external environment (for example an attacker) using the channel c'. Assuming that the action is $c'?^{\mathrm{L}} d_3\{\checkmark\}$, meaning that a new constant d_3 is injected in the system, we arrive at the configuration

$$\begin{array}{ll}
\mathsf{define} & A_1(x) \triangleq \cdots \\
& A_2(x) \triangleq \cdots \\
\mathsf{in} & \langle \mathrm{H}, A_1(\mathsf{none})[\mathsf{some}(\checkmark)/z_1; \mathsf{some}(d_3)/x_1] \\
& \quad \mid A_2(\mathsf{none})[\mathsf{some}(d_1)/x_1; \mathsf{none}/z_2]\rangle \\
\mathsf{using} & c, c', \checkmark, d_1, d_2, d_3
\end{array}$$

so now both processes are ready to unfold. □

In summary, if we admit only steps of the form

$$S \xrightarrow{\tau} S' \qquad \text{or} \qquad S \xrightarrow{c_1 \dagger^\ell c_2\{c\}} S'$$

we obtain a semantics corresponding to a closed environment, whereas if we also admit steps of the form

$$S \xrightarrow{c_1 !^\ell c_2\{c\}} S' \qquad \text{or} \qquad S \xrightarrow{c_1 ?^\ell c_2\{c\}} S'$$

we obtain a semantics corresponding to an open environment.

4 Safety and Security Type System

The aim of the type systems is to propagate information about the trust levels in order not only to validate that the constraints imposed on trust levels are satisfied but also that the subprocesses that are considered unreachable are indeed

Table 5. Type system for terms, expressions and (extended) binders

$$\Gamma \vdash y : \Gamma(y) \qquad \frac{\Gamma \vdash t : \ell}{\Gamma \vdash \mathsf{endorse}^{\ell_2}_{\ell_1}(t) : \ell_2} \text{ if } \ell \sqsupseteq \ell_1$$

$$\Gamma \vdash c : \top \qquad \frac{\Gamma \vdash t_1 : \ell_1 \quad \cdots \quad \Gamma \vdash t_n : \ell_n}{\Gamma \vdash f(t_1, \cdots, t_n) : \ell_1 \sqcap \cdots \sqcap \ell_n}$$

$$\Gamma \vdash x \triangleright \Gamma(x), x \qquad \frac{\Gamma \vdash t : \ell}{\Gamma \vdash \mathsf{some}(t) \triangleright \ell, \mathsf{tt}} \qquad \Gamma \vdash \mathsf{none} \triangleright \top, \mathsf{ff}$$

$$\frac{\Gamma \vdash t_1 : \ell_1 \qquad \Gamma \vdash t_2 : \ell_2}{\ell', \Gamma \vdash t_1 !^{\ell} t_2\{x\} \blacktriangleright [x \mapsto \ell], x} \text{ if } \ell_2 \sqsupseteq \ell \text{ and } \ell_1 \sqsupseteq \ell \text{ and } \ell' \sqsupseteq \ell$$

$$\frac{\Gamma \vdash t_1 : \ell_1 \qquad \Gamma \vdash t_2 : \ell_2}{\ell', \Gamma \vdash t_1 ?^{\ell} x\{t_2\} \blacktriangleright [x \mapsto \ell], x} \text{ if } \ell_2 \sqsupseteq \ell \text{ and } \ell_1 \sqsupseteq \ell \text{ and } \ell' \sqsupseteq \ell$$

$$\frac{\ell', \Gamma \vdash b_1 \blacktriangleright \Gamma_1, \varphi_1 \quad \cdots \quad \ell', \Gamma \vdash b_n \blacktriangleright \Gamma_n, \varphi_n}{\ell', \Gamma \vdash \&_q(b_1, \cdots, b_n) \blacktriangleright \Gamma_1 \circ \cdots \circ \Gamma_n, [\![q]\!](\varphi_1, \cdots, \varphi_n)}$$

$$\text{if } \mathsf{bv}(b_i) \cap \mathsf{bv}(b_j) = \emptyset \text{ for } i \neq j \text{ and } \mathsf{bv}(b_i) \cap \mathsf{fv}(b_j) = \emptyset \text{ for all } i, j$$

$$\frac{\ell_2, \Gamma \vdash b \blacktriangleright \Gamma_b, \varphi_b}{\ell', \Gamma \vdash \mathsf{assert}^{\ell_2}_{\ell_1}(b) \blacktriangleright \Gamma_b, \varphi_b} \text{ if } \ell' \sqsupseteq \ell_1$$

$$\frac{\Gamma \vdash t : \ell''}{\ell', \Gamma \vdash [\mathsf{some}(t)/x : \ell] \blacktriangleright [x \mapsto \ell], \mathsf{tt}} \text{ if } \ell' \sqsupseteq \ell \text{ and } \ell'' \sqsupseteq \ell$$

unreachable. To do that we shall make use of a *type environment* Γ assigning trust levels to variables (denoted x and y above) and process names (denoted A above). In Table 5 we specify typing judgements for terms, expressions and binders and in Table 6 we specify the judgements for processes and configurations.

The typing judgement for *terms* takes the form $\Gamma \vdash t : \ell$ and indicates the well-typedness of the term t in the type environment Γ and states that t has the trust level $\ell \in \mathcal{L}^{\top}$. The trust level $\top$ is assigned to all constants as they are not affected by the communications and for function application we use the greatest lower bound operation to determine the trust level of the composite construct. For the $\mathsf{endorse}^{\ell_2}_{\ell_1}(\cdot)$ construct we insist that the data being endorsed is at least at the trust level of ℓ_1 and it is then lifted to that of ℓ_2.

The typing judgement for *expressions* takes the form $\Gamma \vdash e \triangleright \ell, \varphi$. As for the terms, this judgement indicates the well-typedness of the expression e in the type environment Γ and states that e has trust level $\ell \in \mathcal{L}^{\top}$. Furthermore, the logical formula φ indicates whether or not e is actually containing data; here we assume that the status of a free expression variable x is given by a boolean variable (that for the sake of simplicity also is named x).

Table 6. Type system for processes, configurations and systems

$$\frac{\varphi, \ell', \Gamma \vdash P}{\varphi, \ell', \Gamma \vdash (\nu c)\, P} \qquad \frac{\varphi, \ell', \Gamma \vdash P_1 \quad \varphi, \ell', \Gamma \vdash P_2}{\varphi, \ell', \Gamma \vdash P_1 \mid P_2} \qquad \varphi, \ell', \Gamma \vdash \mathsf{0}$$

$$\frac{\ell', \Gamma \vdash b \blacktriangleright \Gamma_b, \varphi_b \quad \varphi', \ell', \Gamma[\Gamma_b] \vdash P}{\varphi, \ell', \Gamma \vdash b.P} \text{ if } (\exists \mathsf{bv}(b).\varphi) \wedge \varphi_b \;\Rightarrow\; \varphi'$$

$$\frac{\Gamma \vdash e \triangleright \ell_e, \varphi_e \quad \varphi \wedge \varphi_e, \ell, \Gamma[y \mapsto \ell] \vdash P_1 \quad \varphi \wedge \neg\varphi_e, \ell, \Gamma \vdash P_2}{\varphi, \ell', \Gamma \vdash \mathsf{case}^{\ell}\ e\ \mathsf{of}\ \mathsf{some}(y) \colon P_1\ \mathsf{else}\ P_2} \text{ if } \ell_e \sqsupseteq \ell \text{ and } \ell' \sqsupseteq \ell$$

$$\frac{\Gamma \vdash e \triangleright \ell_e, \varphi_e}{\varphi, \ell', \Gamma \vdash A(e)} \text{ if } (\ell_x, \ell') \in \Gamma(A) \text{ and } \ell_e \sqsupseteq \ell_x \qquad \varphi, \ell', \Gamma \vdash \mathsf{dummy}(P) \quad \text{if } \mathsf{unsat}(\varphi)$$

$$\frac{\Gamma \vdash Q}{\Gamma \vdash (\nu c)\, Q} \qquad \frac{\Gamma \vdash Q_1 \quad \Gamma \vdash Q_2}{\Gamma \vdash Q_1 \mid Q_2} \qquad \frac{\mathsf{tt}, \ell, \Gamma \vdash P}{\Gamma \vdash \langle \ell, P \rangle} \qquad \Gamma \vdash \mathsf{0}$$

$$\frac{\begin{array}{l}\forall (\ell', \ell) \in \Gamma(A_1) : \ \mathsf{tt}, \ell, \Gamma[x_1 \mapsto \ell'] \vdash P_1 \\ \vdots \\ \forall (\ell', \ell) \in \Gamma(A_n) : \ \mathsf{tt}, \ell, \Gamma[x_n \mapsto \ell'] \vdash P_n \qquad \text{if } \mathsf{dom}(\Gamma) = \{A_1, \cdots, A_n\} \\ \Gamma \vdash Q_*\end{array}}{\Gamma \vdash \mathsf{define}\ A_1(x_1) \triangleq P_1\ \cdots\ A_n(x_n) \triangleq P_n\ \mathsf{in}\ Q_*\ \mathsf{using}\ c_1, \cdots, c_k}$$

The typing judgement for *binders* takes the form $\ell, \Gamma \vdash b \blacktriangleright \Gamma', \varphi'$ and indicates the well-typedness of the binder b in the type environment Γ and in a context specified by the trust level $\ell \in \mathcal{L}$. Furthermore the judgement expresses that the binder produces the new environment Γ' and the logical formula φ' indicates whether or not the binder has been fully evaluated. The clauses for output and input binders impose constraints on the trust levels: for output we require that the data being sent is at least at the intended trust level, that the trust level of the channel is at least at the intended trust level, and we also require that the action happens in a sufficiently high context. Similar conditions are imposed by the clause for input. In both cases the bound variable is assigned the intended trust level. In the clause for composite binders the side condition ensures that type environments constructed by the subbinders have disjoint domains and hence can be composed and we use the semantics $[\![q]\!]$ of the quality predicate q to compose the logical formulae obtained from the subbinders. Finally, in the clause for the construct $\mathsf{assert}^{\ell_2}_{\ell_1}(\cdot)$ we require that the trust level ℓ' of the context is at least ℓ_1 and the subbinder is then type checked in the context ℓ_2.

The typing judgement for *processes* is specified in Table 6 and takes the form $\varphi, \ell, \Gamma \vdash P$ and indicates the well-typedness of the process P in the type

environment Γ, at trust level $\ell \in \mathcal{L}$, and under the assumption that data is available as specified by the logical formula φ. The clause for prefixing will determine a local type environment Γ_b and an availability formula φ_b for the binder and then use that to describe the context in which the continuation process has to be type checked. The reachability formula for the continuation is $(\exists \mathsf{bv}(b).\varphi) \wedge \varphi_b$ which means existentially quantifying over all bound variables of b in the formula φ and taking the conjunction with φ_b. It is always safe to replace a reachability formula by a logical consequence (since $\varphi_1 \Rightarrow \varphi_2$ ensures $\mathsf{unsat}(\varphi_2) \Rightarrow \mathsf{unsat}(\varphi_1)$) and we do so in Table 6 in order to ensure that well-typedness is preserved under evaluation (see Theorem 1 below).

For the case construct we use the intended trust level to type check the two subprocesses; in both cases the availability formula is modified to take care of the expression; finally, we check that the trust level of the expression and of the context both dominate the intended trust level. In the clause for process calls we use that the type environment additionally maps a process names A to sets of pairs (ℓ_x, ℓ') indicating that for an actual parameter of trust level ℓ_e such that $\ell_e \sqsupseteq \ell_x$, the body of the process definition is typable at trust level ℓ'. Finally, the construct $\mathsf{dummy}(P)$ is typable if it is not reachable, that is, if the availability formula is unsatisfiable.

The typing judgement for *configurations* takes the form $\Gamma \vdash Q$ and indicates the well-typedness of the configuration Q in the type environment Γ obtained from the overall system. Its definition follows largely the pattern of processes. The final clause in Table 6 expresses when a system is well-typed; it implicitly expresses a fixed point property defining the type environment Γ for the process names.

Example 3. It is easy to check that the system of Example 1 is well-typed using the initial type environment Γ given by $\Gamma(A_1) = \{(\mathrm{H}, \mathrm{H})\}$ and $\Gamma(A_2) = \{(\mathrm{L}, \mathrm{H})\}$. In particular we obtain

$$\mathrm{H}, \Gamma[x \mapsto \mathrm{H}] \vdash \&_\forall(c!^{\mathrm{H}} d_1\{z_1\}, c'?^{\mathrm{L}} x_1\{\checkmark\}) \blacktriangleright [z_1 \mapsto \mathrm{H}; x_1 \mapsto \mathrm{L}], z_1 \wedge x_1$$

$$\mathrm{H}, \Gamma[x \mapsto \mathrm{L}] \vdash \&_\exists(c?^{\mathrm{H}} x_2\{\checkmark\}, c'!^{\mathrm{L}} d_2\{z_2\}) \blacktriangleright [x_2 \mapsto \mathrm{H}; z_2 \mapsto \mathrm{L}], x_1 \vee z_2$$

for the two binders of the processes. □

5 Worked Example

We shall consider a scenario where the cd-player of a car is used not only for playing music but also for updating the functionality of the car. The overall functionality of the controller of the cd-player is as following:

- It will accept input from the driver of the car (on channel driver) and the cd (on channel cd) and will then play the music on the cd (modelled as output on channel play).
- It will also accept input from the service center (on channel service) and the cd and will then start updating the software of the car (modelled as output on the channel update); the controller holds information about the current version of the software and it will also be updated.

Table 7. The controller

$$
\begin{array}{lll}
\mathsf{ctrl}(x_v) \triangleq \&_\forall(\mathsf{cd}?^{\mathrm{L}}x_c\{\checkmark\}, \&_\exists(\mathsf{driver}?^{\mathrm{M}}x_d\{\checkmark\}, \mathsf{service}?^{\mathrm{H}}x_s\{\checkmark\})) & & (1)\\
\quad \mathsf{case}^{\mathrm{H}}\ x_s\ \mathsf{of\ some}(y_s): & & (2)\\
\quad\quad \mathsf{case}^{\mathrm{M}}\ x_d\ \mathsf{of\ some}(y_d): & & (3)\\
\quad\quad\quad \mathsf{case}^{\mathrm{L}}\ x_c\ \mathsf{of\ some}(y_c): & & (4)\\
\quad\quad\quad\quad \&_\exists(\mathsf{assert}^{\mathrm{H}}_{\mathrm{L}}(\mathsf{update}!^{\mathrm{H}}\mathsf{f}(y_s, \mathsf{endorse}^{\mathrm{H}}_{\mathrm{L}}(y_c))\{z_u\}), & & (5)\\
\quad\quad\quad\quad\quad \mathsf{play}!^{\mathrm{L}}\mathsf{g}(y_d, y_c)\{z_p\}). & & (6)\\
\quad\quad\quad\quad\quad \mathsf{case}^{\mathrm{H}}\ z_u\ \mathsf{of\ some}(y_u): & & (7)\\
\quad\quad\quad\quad\quad\quad \mathsf{ctrl}(\mathsf{some}(\mathsf{n}(\mathsf{endorse}^{\mathrm{H}}_{\mathrm{L}}(y_c)))) & & (8)\\
\quad\quad\quad\quad\quad \mathsf{else\ ctrl}(x_v)) & & (9)\\
\quad\quad\quad \mathsf{else\ dummy}(0) & & (10)\\
\quad\quad \mathsf{else\ case}^{\mathrm{L}}\ x_c\ \mathsf{of\ some}(y_c): & & (11)\\
\quad\quad\quad\quad \mathsf{assert}^{\mathrm{H}}_{\mathrm{L}}(\mathsf{update}!^{\mathrm{H}}\mathsf{f}(y_s, \mathsf{endorse}^{\mathrm{H}}_{\mathrm{L}}(y_c))\{z_u\}). & & (12)\\
\quad\quad\quad\quad\quad \mathsf{ctrl}(\mathsf{some}(\mathsf{n}(\mathsf{endorse}^{\mathrm{H}}_{\mathrm{L}}(y_c)))) & & (13)\\
\quad\quad\quad \mathsf{else\ dummy}(0) & & (14)\\
\quad\quad \mathsf{else\ case}^{\mathrm{M}}\ x_d\ \mathsf{of\ some}(y_d): & & (15)\\
\quad\quad\quad\quad \mathsf{case}^{\mathrm{L}}\ x_c\ \mathsf{of\ some}(y_c): & & (16)\\
\quad\quad\quad\quad\quad \mathsf{play}!^{\mathrm{L}}\mathsf{g}(y_d, y_c)\{z_p\}. & & (17)\\
\quad\quad\quad\quad\quad\quad \mathsf{ctrl}(x_v) & & (18)\\
\quad\quad\quad\quad \mathsf{else\ dummy}(0) & & (19)\\
\quad\quad\quad \mathsf{else\ dummy}(0) & & (20)
\end{array}
$$

Table 7 gives a model of the controller of the cd-player in the Quality Calculus. We assume that we have a lattice with three trust levels, H, M and L ordered so that $\mathrm{L} \sqsubseteq \mathrm{M} \sqsubseteq \mathrm{H}$. The binder in line (1) requires that input must be received from the cd-player and either the driver or the service center. Actually it is possible that all three inputs are received and lines (5-9) takes care of this situation; lines (12-13) handle the case where input is received from the service center and the cd-player whereas lines (17-18) take care of the case where input is received from the driver and the cd-player. A number of nested case constructs are used to identify these situations; this also identifies a number of possibilities that indeed cannot arrise, given the choice of binder in line (1), and they are all written as $\mathsf{dummy}(0)$.

The simplest case is when input has been received from the driver and the cd-player; this case is handled in line (17-18) and simply amounts to issuing an output on the channel play and a recursive call of the controller with unchanged parameter. The value being output on play is $\mathsf{g}(y_d, y_c)$; here y_d will have trust level M while y_c will have trust level L so the composite expression will have the trust level L and since the communication has trust level L the type checking will succeed.

Let us next consider the situation where input is received from the service center and the cd-player but not from the driver; this is handled in lines (12-13). In this case we issue an output on the channel update but we cannot just write $\mathsf{update}!^{\mathrm{H}}\mathsf{f}(y_s, y_c)\{z_u\}$ as it will not type check. The reason is that the trust level of the composite expression $\mathsf{f}(y_s, y_c)$ is L (as y_s has trust level H and y_c has trust

level L) and the communication must happen at the trust level H. By using the argument $\mathsf{endorse}^{\mathrm{H}}_{\mathrm{L}}(y_c)$ we lift the trust level of y_c from L to H and the output action will then type check. However, the output occurs as a branch of a case construct that is governed by the variable x_c that has trust level L; therefore we use the $\mathsf{assert}^{\mathrm{H}}_{\mathrm{L}}(\cdot)$ construct to signal that we can pretend to be in a high context and now this part of the process will type check.

Finally let us return to the case where all three inputs are available; this situation is handled in lines (5-9). Here the binder in line (5) captures that we are ready to communicate over the update as well as play channel; the use of the quality predicate $\exists$ means that we continue as soon as one of the outputs have been accepted. In line (7) we then check the acknowledgement from the update channel; if successful we will recurse with an updated parameter and otherwise we recurse with the old parameter.

As already noticed a number of dummy annotations have been inserted reflecting that these parts of the process are unreachable. As an example our type system will generate the following formula for the process in line (10):

$$x_c \wedge (x_d \vee x_s) \wedge x_s \wedge x_d \wedge \neg x_c$$

and it is easy to see that it is unsatisfiable. On the other hand for the process in line (9) we obtain the formula

$$x_c \wedge (x_d \vee x_s) \wedge x_s \wedge x_d \wedge x_c \wedge (z_u \vee z_p) \wedge \neg z_u$$

and clearly it is satisfiable so it would not be correct to replace line (9) with $\mathsf{dummy}(0)$.

6 Theoretical Properties

We are going to establish three theorems for showing the well-behavedness of the type system with respect to the semantics.

Subject reduction. The first correctness statement for the type system is to establish a subject reduction result. Since we have not given the detailed definitions of the semantic relations $t \rhd c$, $e \rhd \mathsf{some}(c)$ and $e \rhd \mathsf{none}$ we will have to assume that they behave in a benign manner in order to establish the results for configurations.

We shall also use that b is well-typed and and hence does not define the same variable twice in order to write $b \xrightarrow{\alpha,x} b'$ to indicate that in the proof of $b \xrightarrow{\alpha} b'$ the variable x was instantiated to some constant of the form $\mathsf{some}(c)$.

Theorem 1. *Assume that*

$$\begin{aligned}
\Gamma \vdash t : \ell \ \wedge \ t \rhd c \quad &\Rightarrow \quad \exists \ell' \sqsupseteq \ell : \ \Gamma \vdash c : \ell' \\
\Gamma \vdash e \rhd \ell, \varphi \ \wedge \ e \rhd \mathsf{some}(c) \quad &\Rightarrow \quad \exists \ell' \sqsupseteq \ell : \ \Gamma \vdash \mathsf{some}(c) \rhd \ell', \varphi \ \wedge \ \varphi \Leftrightarrow \mathsf{tt} \\
\Gamma \vdash e \rhd \ell, \varphi \ \wedge \ e \rhd \mathsf{none} \quad &\Rightarrow \quad \exists \ell' \sqsupseteq \ell : \ \Gamma \vdash \mathsf{none} \rhd \ell', \varphi \ \wedge \ \varphi \Leftrightarrow \mathsf{ff}
\end{aligned}$$

Then

$$\begin{array}{rcl} \ell, \Gamma \vdash b \blacktriangleright \Gamma', \varphi' \ \wedge \ b \xrightarrow{\alpha,x} b' & \Rightarrow & \ell, \Gamma \vdash b' \blacktriangleright \Gamma', \varphi'[\mathsf{tt}/x] \\ \ell, \Gamma \vdash b \blacktriangleright \Gamma', \varphi' \ \wedge \ b ::_v \theta & \Rightarrow & \varphi' \overline{\theta} \Leftrightarrow v \\ \Gamma \vdash Q_1 \ \wedge \ Q_1 \xrightarrow{\alpha} Q_2 & \Rightarrow & \Gamma \vdash Q_2 \\ \Gamma \vdash S_1 \ \wedge \ S_1 \xrightarrow{\alpha} S_2 & \Rightarrow & \Gamma \vdash S_2 \end{array}$$

Here $\overline{\theta}$ is obtained from θ by taking $\overline{\theta}(x) = \mathsf{tt}$ if $\theta(x) = \mathsf{some}(c)$ for some constant c and $\overline{\theta}(x) = \mathsf{ff}$ if $\theta(x) = \mathsf{none}$.

Sketch of proof. The need to replace ℓ by a possibly larger ℓ' is due to the fact that functions are allowed to have arguments that is greater than or equal to the trust level promised as a result; due to evaluation we will typically select or combine a subset of the arguments and hence the greatest lower bound of the result is possibly larger. We refer to Appendix A for more details. □

Safety. It follows from Theorem 1 that any configuration reachable from a well-typed configuration by means of well-typed communications with the environment will itself be well-typed. Hence for the safety result it suffices to show that subprocesses of the form $\mathsf{dummy}(P)$ can never become available for execution and this can be expressed clearly by saying that no well-typed configuration contains a sub-configuration of the form $\langle \ell, \mathsf{dummy}(P) \rangle$. This guarantees that the type system is correct in stating that the details of P are of no consequence because $\mathsf{dummy}(P)$ is not reachable.

Theorem 2. *If $\Gamma \vdash S$ then S does not contain a sub-configuration of the form $\langle \ell, \mathsf{dummy}(P) \rangle$.*

Proof. We proceed by contradiction and suppose that $\Gamma \vdash S$ and that S contains a sub-configuration of the form $\langle \ell, \mathsf{dummy}(P) \rangle$. By the syntax of configurations given in Table 1 and the typing rules for configurations given in Table 6 it is immediate that $\Gamma' \vdash \langle \ell, \mathsf{dummy}(P) \rangle$ for some Γ'. Using the typing rules for configurations once more it is immediate that $\mathsf{tt}, \ell, \Gamma' \vdash \langle \ell, \mathsf{dummy}(P) \rangle$ and hence that $\mathsf{unsat}(\mathsf{tt})$. This establishes the desired contradiction. □

Security. To express the security result we shall fix a given system S of the form

$$\mathsf{define}\ A_1(x_1) \triangleq P_1 \ \cdots \ A_n(x_n) \triangleq P_n \ \mathsf{in}\ Q_* \ \mathsf{using}\ c_1, \cdots, c_k \tag{1}$$

as well as the environment Γ. We shall next develop a non-interference result for showing the correctness of the trust annotations [15,3,10]. It will formalise an attack model where attackers are unknown but well-typed processes that could execute in parallel with the given system.

Let us write $\mathsf{level}(c_1 ?^{\ell} c_2\{c\}) = \ell$, $\mathsf{level}(c_1 !^{\ell} c_2\{c\}) = \ell$, $\mathsf{level}(c_1 \dagger^{\ell} c_2\{c\}) = \ell$, and $\mathsf{level}(\tau) = \bot$ for the level of trust called upon in a given step. Here we are considering $\mathcal{L}^{\top}_{\bot}$ that is the lattice $\mathcal{L}^{\top}$ extended with a new least element $\bot$ merely in order to assign a trust level to τ actions that always satisfies tests of the form $\mathsf{level}(\tau) \sqsubseteq \ell$ for $\ell \in \mathcal{L}^{\top}$.

Define $S \xrightarrow{\sqsubseteq\lambda} S'$ to mean $S \xrightarrow{\alpha} S'$ and $\mathsf{level}(\alpha) \sqsubseteq \lambda$ and write $S \stackrel{\sqsubseteq\lambda}{\Longrightarrow} S'$ for the reflexive transitive closure of this relation. Define $S \xrightarrow{\alpha,\not\sqsubseteq\lambda} S'$ to mean $S \xrightarrow{\alpha} S'$ and $\mathsf{level}(\alpha) \not\sqsubseteq \lambda$ and write $S \stackrel{\alpha,\not\sqsubseteq\lambda}{\Longrightarrow} S'$ for $\exists S'' : S \stackrel{\sqsubseteq\lambda}{\Longrightarrow} S'' \wedge S'' \xrightarrow{\alpha,\not\sqsubseteq\lambda} S'$.

Definition 1. *A relation R is a λ-bisimulation provided that it is symmetric and that whenever $S_1\ R\ S_2$ we have that*

1. $\Gamma \vdash S_1$
2. (a) *if* $S_1 \stackrel{\sqsubseteq\lambda}{\Longrightarrow} S_1'$ *and* $S_2 \stackrel{\sqsubseteq\lambda}{\Longrightarrow} S_2'$
   (b) *and* $S_1' \xrightarrow{\alpha,\not\sqsubseteq\lambda} S_1''$
   (c) *then it possible to find* $S_2' \stackrel{\alpha,\not\sqsubseteq\lambda}{\Longrightarrow} S_2''$ *such that* $S_1''\ R\ S_2''$.

The notion of λ-bisimulation (in particular λ-bisimilarity defined below) embodies the following idea. Consider two systems S_1 and S_2 that are related by some λ-bisimulation. We can let them execute freely with the environment as long as we stay at a trust level that is dominated by $\lambda \in \mathcal{L}$. If one of the resulting systems (say S_1') can then perform an action at a trust level not dominated by λ then so can the other system (say S_2') — possibly after having performed additional steps at a trust level dominated by λ in order to reach the step of interest. The fact that the resulting systems (S_1'' and S_2'') are also related by the λ-bisimulation means that the above process can be repeated as often as possible.

It is immediate to show that the union of all λ-bisimulations is itself a λ-bisimulation; this motivates the following definition.

Definition 2. *λ-bisimilarity $\underset{\lambda}{\approx}$ is defined as the largest λ-bisimulation.*

We can now express the non-interference result that communications at trust level dominated by λ cannot influence communications at trust levels not dominated by λ.

Theorem 3. *If the system S of* (1) *satisfies $\Gamma \vdash S$ and has no occurrences of* $\mathsf{endorse}_{\cdot}(\cdot)$ *or* $\mathsf{assert}_{\cdot}(\cdot)$ *then $S \underset{\lambda}{\approx} S$ for all $\lambda \in \mathcal{L}$.*

Sketch of proof. It is crucial for the proof that the Quality Calculus does not contain any nondeterministic choices and that these cannot be encoded because of our consideration of an open system semantics. We refer to Appendix B for more details. □

7 Conclusion

Safety is the traditional concern when designing embedded systems but when embedded systems become parts of cyber-physical systems the open nature of communications necessitates that security becomes an equal player. The security concerns of highest importance are trust (or integrity or authenticity) and

availability. It is then up to the developer to design a system that strikes the proper balance between safety and security.

To assist in the design of such systems we have devised the Quality Calculus. At the outset [12] this is a calculus in the π-calculus family of languages [11] that allows to express *quality* predicates on the inputs that must have succeeded in order for computation to progress. In this paper the Quality Calculus has been extended in a number of ways:

- We performed a symmetric treatment (also see [13]) of input and output as regards the quality predicates.
- We incorporated trust levels to indicate that some communications may be more trustworthy than others, for example because of the strength of cryptography used in securing them.
- We introduced language primitives for temporarily allowing to violate the demands on the explicit flow of information (by means of explicit endorsement) as well as the implicit flow of information (by means of explict assertions); in both cases it is incumbent upon the designer to take responsibility for the resulting violations of the information flow.
- We included a language primitive for expressing that certain parts of the processes are not reachable and that therefore the one can dispense with proper "error" handling; the mechanism to be used by the designer for achieving this goal amounts to having default or substitute data available in case expected data is not received from other distributed components.

The main technical contribution of the present paper was the development of a type system that simultanously enforced the safety and security policies. The safety component was realised by a logical formula indicating the reachability of the various subprocesses; a subprocesses is deemed to be unreachable if the logical formula is unsatisfiable and this is a correct over-approximation of reachability. The security component was realised by a type environment mapping entities to trust levels and ensuring that (in the absence of endorsement and assertion) that highly trusted communication cannot be influenced by less trusted communication and thereby securing communication at the the higher levels against any attacker able to penetrate the cryptographic defences at lower levels. This was illustrated on a worked example taken from the automotive sector and we concluded by stating the formal results expressing the correctness of the type system.

In future work we would like to extend the present approach with more quantitative considerations of lack of availability perhaps based on stochastic models of the robustness of communication.

Acknowledgement. The research was supported by MT-LAB, a VKR Centre of Excellence for the Modelling of Information Technology, and IDEA4CPS, supported by the Danish Foundation for Basic Research, and was motivated by the considerations of SESAMO, an European Artemis project.

References

1. Amadio, R.M.: An asynchronous model of locality, failure and process mobility. In: Garlan, D., Le Métayer, D. (eds.) COORDINATION 1997. LNCS, vol. 1282, pp. 374–391. Springer, Heidelberg (1997)
2. Berger, M., Honda, K.: The two-phase commitment protocol in an extended pi-calculus. Electr. Notes Theor. Comput. Sci. 39(1), 21–46 (2000)
3. Birgisson, A., Russo, A., Sabelfeld, A.: Unifying facets of information integrity. In: Jha, S., Mathuria, A. (eds.) ICISS 2010. LNCS, vol. 6503, pp. 48–65. Springer, Heidelberg (2010)
4. de Moura, L.M., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM 54(9), 69–77 (2011)
5. De Nicola, R., Gorla, D., Pugliese, R.: Basic observables for a calculus for global computing. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1226–1238. Springer, Heidelberg (2005)
6. Francalanza, A., Hennessy, M.: A Theory for Observational Fault Tolerance. J. Log. Algebr. Program. 73(1-2), 22–50 (2007)
7. Gollmann, D.: Computer Security, 3rd edn. Wiley (2011)
8. Lee, E.A.: Cyber physical systems: Design challenges. In: 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008), pp. 363–369. IEEE Computer Society (2008)
9. Malik, S., Zhang, L.: Boolean satisfiability from theoretical hardness to practical success. Commun. ACM 52(8), 76–82 (2009)
10. Mantel, H., Sands, D., Sudbrock, H.: Assumptions and guarantees for compositional noninterference. In: Proceedings of the 24th IEEE Computer Security Foundations Symposium, pp. 218–232. IEEE Computer Society (2011)
11. Milner, R.: Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press (1999)
12. Nielson, H.R., Nielson, F., Vigo, R.: A calculus for quality. In: Păsăreanu, C.S., Salaün, G. (eds.) FACS 2012. LNCS, vol. 7684, pp. 188–204. Springer, Heidelberg (2013)
13. Parrow, J., Victor, B.: The fusion calculus: Expressiveness and symmetry in mobile processes. In: Proc. IEEE Symp. on Logic in Computer Science, pp. 176–185 (1998)
14. Riely, J., Hennessy, M.: Distributed processes and location failures. Theor. Comput. Sci. 266(1-2), 693–735 (2001)
15. Sabelfeld, A., Myers, A.C.: A model for delimited information release. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) ISSS 2003. LNCS, vol. 3233, pp. 174–191. Springer, Heidelberg (2004)
16. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security 4(2/3), 167–188 (1996)

A Proof of Theorem 1

From the assumptions of Theorem 1 we shall now establish a series of lemmata from which the four results of Theorem 1 follows.

The first two results focus on the binders; in both cases the proofs are by structural induction:

Lemma 1. *Assume* $\ell', \Gamma \vdash b \blacktriangleright \Gamma', \varphi$ *and* $b \xrightarrow{\alpha,x} b'$. *Then* $\ell', \Gamma \vdash b' \blacktriangleright \Gamma', \varphi[\mathsf{tt}/x]$.

Lemma 2. *Assume* $\ell', \Gamma \vdash b \blacktriangleright \Gamma', \varphi$ *and* $b ::_v \theta$. *Then* $v \Leftrightarrow \varphi\overline{\theta}$.

To prove the third part of the Subject Reduction result we shall first establish three classical lemmata. The first is a substitution result and for this we define $\Gamma \vdash \theta$ to mean that $\Gamma(\theta(x)) \sqsupseteq \Gamma(x)$ for all x in the domain of θ. Then we have:

Lemma 3. *Assume that* $\varphi, \ell, \Gamma \vdash P$ *and that* $\Gamma \vdash \theta$. *Then* $\varphi\overline{\theta}, \ell, \Gamma \vdash P\theta$.

The proof is by a straightforward structural induction. It relies on substitution results for expressions and binders stating that if $\Gamma \vdash e \triangleright \ell, \varphi$ then $\Gamma \vdash e\theta \triangleright \ell, \varphi\overline{\theta}$ and if $\ell, \Gamma \vdash b \blacktriangleright \Gamma_b, \varphi_b$ then $\ell, \Gamma \vdash b\theta \blacktriangleright \Gamma_b, \varphi_b$; in the latter case we assume that $\mathsf{bv}(b) \cap \mathsf{dom}(\theta) = \emptyset$ (this can easily be satisfied by alpha renaming the bound variables).

The next lemma allows us to restrict the type environment to the variables of interest and this is followed by a lemma stating that structural congruent configurations have the same typing properties; in the first case the proof is by a straightforward structural induction and in the second case it follows from an induction on the definition of the congruence relation.

Lemma 4. *Assume that* $\varphi, \ell, \Gamma \vdash P$ *and that* $\Gamma'(x) = \Gamma(x)$ *for all* $x \in \mathsf{fv}(P)$. *Then* $\varphi, \ell, \Gamma' \vdash P$.

Lemma 5. *Assume* $Q_1 \equiv Q_2$. *Then* $\Gamma \vdash Q_1$ *if and only if* $\Gamma \vdash Q_2$.

Now the remaining results of Theorem 1 follow from the following subject reduction result for configurations:

Lemma 6. *Assume* $\Gamma \vdash Q_1$ *and* $Q_1 \xrightarrow{\alpha} Q_2$. *Then* $\Gamma \vdash Q_2$.

The proof is by induction on the inference of the semantics and relies on the results established above. The most interesting cases are those of communication and here the results of Lemma 1, 2 and 3 are combined to give the result. To handle the case construct we rely on Lemma 3 and 4 and Lemma 5 is used in the case where the congruence is embedded in the semantics. We omit the details.

B Proof of Theorem 3

From the assumptions of Theorem 3 we have that the system S of the form

$$\mathsf{define}\ A_1(x_1) \triangleq P_1\ \cdots\ A_n(x_n) \triangleq P_n\ \mathsf{in}\ Q_*\ \mathsf{using}\ c_1, \cdots, c_k$$

satisfies $\Gamma \vdash S$ and has no occurrences of $\mathsf{endorse}_{\cdot}(\cdot)$ or $\mathsf{assert}_{\cdot}(\cdot)$. We next fix $\lambda \in \mathcal{L}$ and need to show that $S \underset{\lambda}{\approx} S$. To do so it suffices to construct a relation R such that $S\ R\ S$ and R is a λ-bisimulation.

For the construction we shall make use of an auxiliary function $\mathsf{close}_\lambda(\cdots)$ defined as follows:

$$\mathsf{close}_\lambda(R) = \{\ (S_1', S_2')\ \mid\ S_1 \stackrel{\sqsubseteq\lambda}{\Longrightarrow} S_1'\ \wedge\ S_2 \stackrel{\sqsubseteq\lambda}{\Longrightarrow} S_2'\ \wedge\ (S_1, S_2) \in R\ \}$$

We next define $R_0 = \{(S, S)\}$ and

$$\begin{aligned} R_{i+1} &= \{\ (S_1'', S_2'')\ \mid\ S_1' \stackrel{\alpha,\not\sqsubseteq\lambda}{\longrightarrow} S_1''\ \wedge\ S_2' \stackrel{\alpha,\not\sqsubseteq\lambda}{\longrightarrow} S_2''\ \wedge\ (S_1', S_2') \in \mathsf{close}_\lambda(R_i)\ \} \\ &= \{\ (S_1'', S_2'')\ \mid\ S_1' \stackrel{\alpha,\not\sqsubseteq\lambda}{\longrightarrow} S_1''\ \wedge\ S_2' \stackrel{\alpha,\not\sqsubseteq\lambda}{\Longrightarrow} S_2''\ \wedge\ (S_1', S_2') \in \mathsf{close}_\lambda(R_i)\ \} \end{aligned}$$

It is now immediate that $R = \bigcup_i R_i$ is a relation such that $S\ R\ S$.

It remains to show that R is a λ-bisimulation. It is immediate that R is a symmetric relation and the condition that $\Gamma \vdash S_1$ whenever $S_1\ R\ S_2$ follows from Theorem 1 and the assumption that $\Gamma \vdash S$. The remaining condition is a consequence of a few lemmas that are expressed in terms of two auxiliary functions.

Define

$$\mathsf{Exp}_\lambda(S') = \{\alpha \mid \exists S'' :\ S' \stackrel{\alpha,\not\sqsubseteq\lambda}{\longrightarrow} S''\}$$

and

$$\mathsf{Reach}_\lambda(S') = \{\alpha \mid \exists S'' :\ S' \stackrel{\alpha,\not\sqsubseteq\lambda}{\Longrightarrow} S''\}$$

Lemma 7. *For all S' we have $\mathsf{Exp}_\lambda(S') \subseteq \mathsf{Reach}_\lambda(S'')$.*

Lemma 8. *If $S' \stackrel{\sqsubseteq\lambda}{\Longrightarrow} S''$ then $\mathsf{Exp}_\lambda(S') \subseteq \mathsf{Exp}_\lambda(S'')$ and $\mathsf{Reach}_\lambda(S') = \mathsf{Reach}_\lambda(S'')$.*

The proof relies on the absence of nondeterministic choices in the Quality Calculus and that any attempts to encode them using the open semantics would influence the trust level.

Lemma 9. *For all S' there exists S'' such that $S' \stackrel{\sqsubseteq\lambda}{\Longrightarrow} S''$ and $\mathsf{Exp}_\lambda(S') \subseteq \mathsf{Reach}_\lambda(S') = \mathsf{Exp}_\lambda(S'') = \mathsf{Reach}_\lambda(S'')$.*

Invariants Synthesis over a Combined Domain for Automated Program Verification

Shengchao Qin[1,3], Guanhua He[1], Wei-Ngan Chin[2], and Hongli Yang[3]

[1] Teesside University, Middlesbrough TS1 3BA, UK
[2] National University of Singapore, Singapore
[3] Beijing University of Technology
{S.Qin,G.He}@tees.ac.uk, chinwn@comp.nus.edu.sg, yhl.yang@gmail.com

Abstract. Program invariants such as loop invariants and method specifications (a.k.a. procedural summaries) are key components in program verification. Such invariants are usually manually specified by users before passed as inputs to a program verifier. The process of manually annotating programs with such invariants is tedious and error-prone and can significantly hinder the level of automation in program verification. Although invariant synthesis techniques have made noticeable progress in reducing the burden of user annotations; when it comes to automated verification of memory safety and functional correctness for heap-manipulating programs, it remains a rather challenging task to discover program specifications and invariants automatically, due to the complexity of aliasing and mutability of data structures.

In this paper, we present invariant synthesis algorithms for the following scenarios: i) to synthesise a missing loop invariant, ii) to refine given pre/post shape templates to complete pre/post-conditions, iii) to infer a missing precondition, iv) to calculate a missing postcondition, given a precondition. The proposed analyses are based on abstract interpretation and are built over an abstract domain combining separation, numerical and multi-set (bag) information. Our inference mechanisms are equipped with newly designed abstraction, join, widening and abduction operations. Initial prototypical experiments have shown that they are viable and powerful enough to discover interesting useful invariants for non-trivial programs.

1 Introduction

Due to the complexity of software, program errors/bugs may be hidden deeply and may not be exposed solely by testing. Formal verification, on the other hand, can guarantee a higher level correctness. Although research on formal verification has a long history, dating back to the 1960s, it remains a challenging problem to automatically verify (sizeable) programs written in mainstream imperative languages such as C, C++, C# and Java.

Over the last two decades, many verification tools have been developed to help programmers to verify their programs. A common observation made for most of these verification tools [3,2,26,1,21,15] is that they often rely critically on users

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 304–325, 2013.

to manually supply program invariants such as method pre/post-conditions and loop invariants, which can be error-prone, cumbersome, and can significantly hinder the level of automation and the applicability of the tools. Static analysis techniques that can help synthesise program invariants are crucial for the viability and scalability of many verification tools and have been well sought after. Nevertheless, it has been a challenging problem to construct effective program analyses to synthesise program invariants automatically. This is especially so when it comes to automated verification of memory safety and functional correctness for heap-manipulating programs, due to the complexity of aliasing and mutability of data structures under manipulation.

In automatic program invariant synthesis, certain kinds of program properties have been well explored over the last decades, such as numerical properties in linear abstract domains, and structural (shape) properties for list-manipulating programs in separation (shape) domains. However, previous works have not yet automatically analysed program properties involving complex mixed domains, particularly for programs with sophisticated data structures and strong invariants involving both structural and pure (numerical and content) information. For example, it is still non-trivial to discover program properties, such as a list becoming sorted during the execution of a program, a binary search tree remaining balanced before and after the execution of a procedure, or the elements of a list remaining unchanged after reversing the list. This difficulty is not only due to sharing and mutability of data structures under manipulation, but is also due to closely intertwined program properties, such as structure-aligned numerical information (length of lists and height of trees), symbolic contents of data structures (bag of values), and relational numerical information (sortedness and balancedness).

In addition to classical shape analyses for invariants synthesis (e.g. [7,13,22,41]), separation logic [20,37] has been applied to analyse shape properties of heap-manipulating programs in recent years [8,14,42,32,35]. These works can automatically infer method specifications in the shape domain. Some other works such as [27,28] also incorporate simple numerical information into their shape domain to allow automated synthesis of properties like data structure size information. However, these previous analyses mainly deal with predesignated data structure properties (such as pointer safety for list segments) with fixed numerical templates (such as list length information).

In our previous work [11], we have developed a verification framework catering for a wider range of properties of heap-manipulating programs, covering memory safety and functional correctness. Our verification system offers an expressive specification mechanism where users can supply their own shape predicates to specify properties involving structural (shape) and pure (numerical and bag) information over the above-mentioned combined domain. In this work, we make another step forward by developing advanced static analyses that can help automatically synthesise program invariants over the complex combined domain. Our proposed invariant synthesis algorithms cover the following different scenarios: i) to synthesise a missing loop invariant, ii) to refine given pre/post shape templates (for a procedure) to complete pre/post-conditions, iii) to synthesise a

missing postcondition, given a precondition; iv) to infer a missing precondition. The proposed analyses are based on abstract interpretation and are built over an abstract domain combining separation, numerical and multi-set (bag) information. Our inference mechanisms are equipped with newly designed abstraction, join, widening and abduction operations. We have implemented the proposed analyses in a prototype system. Initial experiments have shown that our analyses are viable and powerful enough to discover interesting useful invariants for non-trivial programs.

2 Preliminaries

2.1 Separation Logic

Separation logic [37] extends Hoare logic to support reasoning about shared mutable data structures. It provides separation conjunction ($*$) to form formulae like $\mathtt{p}_1 * \mathtt{p}_2$ to assert that two heaps described by $\mathtt{p}_1$ and $\mathtt{p}_2$ are domain-disjoint. In our framework, we allow users to define inductive predicates in separation logic to specify both separation and pure (size and bag) properties of recursive data structures. For example, with a data structure definition for a node in a list `data Node { int val; Node next; }`, one can define a predicate for a list with its content as

$$\begin{array}{l}\mathtt{llB}(\mathtt{root},\mathtt{n},\mathtt{S}) \equiv (\mathtt{root}{=}\mathtt{null}\wedge\mathtt{n}{=}0\wedge\mathtt{S}{=}\emptyset)\vee\\ \quad(\exists \mathtt{v},\mathtt{q},\mathtt{n}_1,\mathtt{S}_1 \cdot \mathtt{root}{\mapsto}\mathtt{Node}(\mathtt{v},\mathtt{q}){*}\mathtt{llB}(\mathtt{q},\mathtt{n}_1,\mathtt{S}_1)\wedge\mathtt{n}_1{=}\mathtt{n}{-}1\wedge\mathtt{S}{=}\mathtt{S}_1{\sqcup}\{\mathtt{v}\})\end{array}$$

The parameter `root` for the predicate `llB` is the root pointer referring to the list. The length and content of the list are denoted resp. by `n` and the bag `S`, and $\sqcup$ indicates multi-set (bag) union.

If one wants to verify a sorting algorithm, they can specify a non-empty sorted list segment as follows:

$$\begin{array}{l}\mathtt{slsB}(\mathtt{root},\mathtt{mi},\mathtt{mx},\mathtt{S},\mathtt{p}) \equiv (\mathtt{root}{\mapsto}\mathtt{Node}(\mathtt{mi},\mathtt{p})\wedge\mathtt{mi}{=}\mathtt{mx}\wedge\mathtt{S}{=}\{\mathtt{mi}\})\vee\\ \quad(\mathtt{root}{\mapsto}\mathtt{Node}(\mathtt{v},\mathtt{q}){*}\mathtt{slsB}(\mathtt{q},\mathtt{m}_1,\mathtt{mx},\mathtt{S}_1,\mathtt{p})\wedge\mathtt{v}{=}\mathtt{mi}\wedge\mathtt{v}{\leq}\mathtt{m}_1\wedge\mathtt{m}_1{\leq}\mathtt{mx}\wedge\mathtt{S}{=}\mathtt{S}_1{\sqcup}\{\mathtt{v}\})\end{array}$$

where it keeps track of the minimum (`mi`) and maximum (`mx`) values in the list as well as the bag of all values (`S`). `p` is the tail of list segment. Note that we use a shortened notation that unbound variables, such as `q`, `v`, $\mathtt{m}_1$ and $\mathtt{S}_1$, are implicitly existentially quantified.

Such predicates play an important role in our analysis as (1) they are used to help specify desired properties about data structures, and (2) they serve as a guide for our analysis to discover desired program specifications.

In our work we make use of the separation logic prover Sleek [29,11] to prove whether one formula Δ' in the combined abstract domain entails another one Δ: $\Delta' \vdash \Delta * \Delta_{\mathtt{R}}$. Along with the proof, Sleek also computes the frame $\Delta_{\mathtt{R}}$ which is useful for our analysis. For instance, by entailment proof

$$\exists \mathtt{y}\cdot \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{vx},\mathtt{y}){*}\mathtt{llB}(\mathtt{y},\mathtt{n},\mathtt{S}) \vdash \mathtt{llB}(\mathtt{x},\mathtt{m},\mathtt{S}_1){*}\Delta_{\mathtt{R}}$$

We can generate the frame $\Delta_{\mathtt{R}}$ as $\mathtt{m}{=}\mathtt{n}{+}1\wedge\mathtt{S}_1{=}\mathtt{S}{\sqcup}\{\mathtt{vx}\}$.

In an earlier work [8,9], a bi-abductive entailment is proposed for the *shape* domain: given two shape formulae $\mathtt{G}, \mathtt{H}$,

$$\mathtt{G} * [\mathtt{A}] \rhd \mathtt{H} * [\mathtt{F}]$$

infers the anti-frame $\mathtt{A}$ and the frame $\mathtt{F}$ along the entailment proof. An example taken from [9] is

$$\mathtt{x}{\mapsto}\mathtt{null} * \mathtt{z}{\mapsto}\mathtt{null} * [\underline{\mathtt{list(y)}}] \rhd \mathtt{list(x)} * \mathtt{list(y)} * [\underline{\mathtt{z}{\mapsto}\mathtt{null}}]$$

where the $\mathtt{list}(\cdot)$ predicate describes acyclic, $\mathtt{null}$-terminated singly-linked lists. In the current work, we will generalise such bi-abductive reasoning to the combined domain (involving shape, user-defined predicates, numerical and bag information). A simple example of the generalised bi-abductive reasoning is

$$\exists \mathtt{y} \cdot \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{vx}, \mathtt{y}) * \mathtt{y}{\mapsto}\mathtt{node}(\mathtt{vy}, \mathtt{null}) * [\underline{\mathtt{A}}] \rhd \mathtt{sllB}(\mathtt{x}, \mathtt{mi}, \mathtt{mx}, \mathtt{S}) * [\underline{\mathtt{F}}]$$

where $\underline{\mathtt{A}} \equiv (\mathtt{vx} \leq \mathtt{vy})$ and $\underline{\mathtt{F}} \equiv (\mathtt{mi}{=}\mathtt{vx} \wedge \mathtt{mx}{=}\mathtt{vy} \wedge \mathtt{S}{=}\{\mathtt{vx}, \mathtt{vy}\})$.

2.2 Language and Abstract Domain

To simplify presentation, we employ a strongly-typed C-like imperative language in Fig. 1 to demonstrate our approach. A program *Prog* written in this language consists of declarations *tdecl*, which can be data type declarations *datat* (e.g. `Node` in Section 2.1), predicate definitions *spred* (e.g. `llB` and `slsB`), as well as method declarations *meth* with method specification *mspec*. The definitions for *spred* and *mspec* are given later in Fig. 2. k^{τ} is a constant of type τ. Our language is expression-oriented, and thus the body of a method (e) is an expression formed by program constructors. d and $d[v]$ represent respectively heap-insensitive and heap sensitive commands. The language allows both call-by-value and call-by-reference method parameters, separated with a semicolon (;).

Prog	::= *tdecl** *meth**	*tdecl* ::= *datat* | *spred*	
datat	::= `data` c { *field** }	*field* ::= $t\ v$	t ::= $c \mid \tau$
meth	::= $t\ mn\ ((t\ v)^*; (t\ v)^*)$ *mspec** $\{e\}$		τ ::= `int` | `bool` | `void`
e	::= $d \mid d[v] \mid v{:=}e \mid e_1; e_2 \mid t\ v;\ e \mid$ `if` $(v)\ e_1$ `else` $e_2 \mid$ `while` $v\ \{e\}$		
d	::= `null` $\mid k^{\tau} \mid v \mid$ `new` $c(v^*) \mid mn(u^*; v^*)$		
$d[v]$	::= $v.f \mid v_1.f{:=}v_2 \mid$ `free`(v)		

Fig. 1. A Core (C-like) Imperative Language

Our specification language (in Fig. 2) allows (user-defined) shape predicates *spred* to specify program properties in our combined domain. Note that such predicates are constructed with disjunctive constraints Φ. We require that the predicates be well-formed [11]. The first parameter of a predicate is the pointer referring to the data structures itself. A conjunctive abstract program state σ has mainly two parts: the heap (shape) part κ in the separation domain and the pure part π in convex polyhedra domain and bag (multi-set) domains, where π consists of γ, ϕ and φ as aliasing, numerical and multi-set information, respectively. $k^{\mathtt{int}}$ is an integer constant. The square symbols like $\sqsubset$, $\sqsubseteq$, $\sqcup$ and $\sqcap$ are multi-set

$$
\begin{array}{lll}
\mathit{spred} & ::= p(\mathtt{root}, v^*) \equiv \Phi \qquad \Phi ::= \bigvee \sigma^* & \sigma ::= \exists v^* \cdot \kappa \wedge \pi \\
\mathit{mspec} & ::= \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} & \\
\Delta & ::= \Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v \cdot \Delta & \\
\kappa & ::= \mathtt{emp} \mid v \mapsto c(v^*) \mid p(v^*) \mid \kappa_1 * \kappa_2 & \pi ::= \gamma \wedge \phi \\
\gamma & ::= v_1 = v_2 \mid v = \mathtt{null} \mid v_1 \neq v_2 \mid v \neq \mathtt{null} \mid \gamma_1 \wedge \gamma_2 & \\
\phi & ::= \varphi \mid b \mid a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg \phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi & \\
b & ::= \mathtt{true} \mid \mathtt{false} \mid v \mid b_1 = b_2 & a ::= s_1 = s_2 \mid s_1 \leq s_2 \\
s & ::= k^{\mathtt{int}} \mid v \mid k^{\mathtt{int}} \times s \mid s_1 + s_2 \mid -s \mid \mathit{max}(s_1, s_2) \mid \mathit{min}(s_1, s_2) \mid |\mathtt{B}| & \\
\varphi & ::= v \in \mathtt{B} \mid \mathtt{B}_1 = \mathtt{B}_2 \mid \mathtt{B}_1 \sqsubset \mathtt{B}_2 \mid \mathtt{B}_1 \sqsubseteq \mathtt{B}_2 \mid \forall v \in \mathtt{B} \cdot \phi \mid \exists v \in \mathtt{B} \cdot \phi & \\
\mathtt{B} & ::= \mathtt{B}_1 \sqcup \mathtt{B}_2 \mid \mathtt{B}_1 \sqcap \mathtt{B}_2 \mid \mathtt{B}_1 - \mathtt{B}_2 \mid \emptyset \mid \{v\} &
\end{array}
$$

Fig. 2. The Specification Language

operators. The set of all σ formulae is denoted as SH (*symbolic heap*). During the symbolic execution, the abstract program state at each program point will be a disjunction of σ's, denoted by Δ. Its set is defined as $\mathcal{P}_{\mathsf{SH}}$. An abstract state Δ can be normalised to the Φ form [11].

Using entailment [11], we define a partial order over these abstract states:

$$\Delta \preceq \Delta' =_{df} \Delta' \vdash \Delta * \Delta_R$$

where Δ_R is the (computed) residue part. We also have an induced lattice over these states as the base of fixpoint calculation for our analysis.

The memory model of our specification formulae can be found in [11]. In our analysis, variables include both program and logical variables.

3 Loop Invariant Synthesis

Loop invariant is the key to prove the correctness of loops. Discovering invariant of loops automatically has been viewed as a challenged task. In this section, we present a solution for synthesising loop invariant in the combined shape and pure domain. Our analysis is based on the framework of abstract interpretation [12] with specifically designed operations (abs, join and widen) over this combined domain.

Our analysis algorithm **LoopInv** is given in Fig. 3. The algorithm takes four input parameters: $\mathcal{T}$ as the program environment with all the method specifications in the program, Δ_{pre} as the precondition for the while loop (i.e. the abstract state before the loop starts), the while loop itself $\mathtt{while}\ b\ \{e\}$, and an upper bound n on the number of shared logical variables we keep during analysis. A shared logical variable (a.k.a. cutpoint) indicates a location which can be reached by two or more program point variables. For example, in formula $\mathtt{sllB}(\mathtt{x}, \mathtt{m}_1, \mathtt{x}_1, \mathtt{S}_1, \mathtt{z}') * \mathtt{sllB}(\mathtt{y}, \mathtt{m}_2, \mathtt{x}_2, \mathtt{S}_2, \mathtt{z}')$, $\mathtt{z}'$ is a cutpoint shared by $\mathtt{x}$ and $\mathtt{y}$. In our analysis, we need the cutpoint to trace the shape of program data structures.

At the beginning of the algorithm, we initialise the iteration variable (i) and two states to begin with (Δ_i and Δ_i'). The starting state of the calculation is Δ_{pre}. Among the two states here, the unprimed version Δ_i denotes the initial state before the i^{th} execution of the loop body, and the primed one Δ_i'

LoopInv($\mathcal{T}$, Δ_{pre}, `while` b $\{e\}$, n)
Local: $i := 0$; $\Delta_i := \Delta_{pre}$; $\Delta'_i := \Delta_i$;
1 **repeat**
2 $i := i + 1$;
3 $\Delta_i := \mathsf{widen}^{\dagger}(\Delta_{i-1}, \mathsf{join}^{\dagger}(\Delta_{i-1}, \Delta'_{i-1}))$;
4 $\Delta'_i := \mathsf{abs}^{\dagger}([\![e]\!]_{\mathcal{T}}(\Delta_i \wedge b))$;
5 **if** $\Delta'_i =$ `false` $\vee$ $\mathsf{cp\_no}(\Delta'_i) > n$ **then return** fail **end if**
6 **until** $\Delta'_i = \Delta'_{i-1}$;
7 **return** Δ'_i

Fig. 3. Loop invariant synthesis algorithm

represents the result state after. At each iteration, the algorithm first joins together the initial state Δ_{i-1} of the previous iteration with the result state Δ'_{i-1} obtained in the previous iteration, and widen it against the state Δ_{i-1} (line 3). Note that this step sets Δ_1 to be Δ_{pre} for the very first iteration. The algorithm then symbolically executes the loop body with the abstract semantics defined in Section 3.1 (line 4), and applies the abstraction operation to the obtained abstract state. If the symbolic execution cannot continue due to a program bug, or if we find our abstraction cannot keep the number of shared logical variables/cutpoints (counted by $\mathsf{cp\_no}$) within a specified bound (n), then a failure is reported (line 5). Otherwise we judge whether a fixpoint is already reached by comparing the current abstract state with the one from the previous iteration (line 6). The fixpoint Δ'_i is returned as the loop invariant. We apply entailment checker to judge whether two formulae are equivalent. If $\Delta_1 \vdash \Delta_2$ and $\Delta_2 \vdash \Delta_1$, we say $\Delta_1 = \Delta_2$.

Since the number of program variables and the set of user-defined predicates are finite, and with a bounded number of cutpoints, our abstract shape domain is finite. This guarantees the termination of our analysis.

In what follows, we elaborate the key techniques of our analysis: the abstract semantics $[\![e]\!]_{\mathcal{T}}\Delta$, the abstraction function abs, and the join and widen operators.

3.1 Abstract Semantics

The abstract semantics is used to execute the loop body symbolically to obtain its post-state during the loop invariant synthesis. Its type is defined as

$$[\![e]\!] \ : \ \mathsf{AllSpec} \rightarrow \mathcal{P}_{\mathsf{SH}} \rightarrow \mathcal{P}_{\mathsf{SH}}$$

where $\mathsf{AllSpec}$ contains all the specifications of all methods (extracted from the program *Prog*). For some expression e, given its precondition, the semantics will calculate the postcondition.

The foundation of the semantics is the basic transition functions from a conjunctive abstract state to a conjunctive or disjunctive abstract state below:

$$\begin{array}{lll} \mathsf{unfold}(x) & : \mathsf{SH} \rightarrow \mathcal{P}_{\mathsf{SH}[x]} & \text{Rearrangement} \\ \mathsf{exec}(d[x]) & : \mathsf{AllSpec} \rightarrow \mathsf{SH}[x] \rightarrow \mathsf{SH} & \text{Heap-sensitive execution} \\ \mathsf{exec}(d) & : \mathsf{AllSpec} \rightarrow \mathsf{SH} \rightarrow \mathsf{SH} & \text{Heap-insensitive execution} \end{array}$$

where $\mathsf{SH}[x]$ denotes the set of conjunctive abstract states in which each element has x exposed as the head of a data node ($c(x, v^*)$), and $\mathcal{P}_{\mathsf{SH}[x]}$ contains all the (disjunctive) abstract states, each of which is composed by such conjunctive states. Here $\mathsf{unfold}(x)$ rearranges the symbolic heap so that the cell referred to by x is exposed for access by heap sensitive commands $d[x]$ via the second transition function $\mathsf{exec}(d[x])$. The third function defined for other (heap insensitive) commands d does not require such exposure of x.

$$\frac{\mathit{isdatat}(c) \quad \sigma \vdash c(x, v^*) * \sigma'}{\mathsf{unfold}(x)\sigma =_{df} \sigma}$$

$$\frac{\mathit{isspred}(c) \quad \sigma \vdash c(x, u^*) * \sigma' \quad c(\mathtt{root}, v^*) \equiv \Phi}{\mathsf{unfold}(x)\sigma =_{df} \sigma' * [x/\mathtt{root}, u^*/v^*]\Phi}$$

The test $\mathit{isdatat}(c)$ returns `true` only if c is a data node and $\mathit{isspred}(c)$ returns `true` only if c is a shape predicate.

The symbolic execution of heap-sensitive commands $d[x]$ (i.e. $x.f_i$, $x.f_i := w$, or $\mathtt{free}(x)$) assumes that the rearrangement $\mathsf{unfold}(x)$ has been done previously:

$$\frac{\mathit{isdatat}(c) \quad \sigma \vdash c(x, v_1, .., v_n) * \sigma'}{\mathsf{exec}(x.f_i)(\mathcal{T})\sigma =_{df} \sigma' * c(x, v_1, .., v_n) \wedge \mathtt{res}{=}v_i}$$

$$\frac{\mathit{isdatat}(c) \quad \sigma \vdash c(x, v_1, .., v_n) * \sigma'}{\mathsf{exec}(x.f_i := w)(\mathcal{T})\sigma =_{df} \sigma' * c(x, v_1, .., v_{i-1}, w, v_{i+1}, .., v_n)}$$

$$\frac{\mathit{isdatat}(c) \quad \sigma \vdash c(x, u^*) * \sigma'}{\mathsf{exec}(\mathtt{free}(x))(\mathcal{T})\sigma =_{df} \sigma'}$$

The symbolic execution rules for heap-insensitive commands are as follows:

$$\mathsf{exec}(k)(\mathcal{T})\sigma =_{df} \sigma \wedge \mathtt{res}{=}k \qquad \frac{\mathit{isdatat}(c)}{\mathsf{exec}(\mathtt{new}\ c(v^*))(\mathcal{T})\sigma =_{df} \sigma * c(\mathtt{res}, v^*)}$$
$$\mathsf{exec}(x)(\mathcal{T})\sigma =_{df} \sigma \wedge \mathtt{res}{=}x$$

$$\frac{\begin{array}{c} t\ mn\ ((t_i\ u_i)_{i=1}^n)\ \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} \in \mathcal{T} \\ \rho = [x_i/u_i]_{i=1}^n \quad \sigma \vdash \rho\Phi_{pr} * \sigma_{fr} \quad \sigma_{po} = \rho\Phi_{po} \end{array}}{\mathsf{exec}(mn(x_1, .., x_n))(\mathcal{T})\sigma =_{df} \sigma_{po} * \sigma_{fr}}$$

The first three rules deal with constant (k), variable (x) and data node creation ($\mathtt{new}\ c(v^*)$), respectively. The keyword `res` indicates the value of the expression. The last rule handles method invocation, and the call site is ensured to meet the precondition of mn, as signified by $\sigma \vdash \rho\Phi_{pr} * \sigma_{fr}$, where σ_{fr} is the frame part. In this case, the execution succeeds and the postcondition of mn ($\rho\Phi_{po}$) is added into the post-state.

A lifting function † is defined to lift unfold's domain to $\mathcal{P}_{\mathsf{SH}}$:

$$\mathsf{unfold}^{\dagger}(x)\bigvee \sigma_i =_{df} \bigvee(\mathsf{unfold}(x)\sigma_i)$$

and this function is overloaded for exec to lift both its domain and range to $\mathcal{P}_{\mathsf{SH}}$:

$$\mathsf{exec}^{\dagger}(d)(\mathcal{T})\bigvee \sigma_i =_{df} \bigvee(\mathsf{exec}(d)(\mathcal{T})\sigma_i)$$

Based on the transition functions above, we can define the abstract semantics for a program command e as follows:

$$\begin{array}{ll}
[\![d[x]]\!]_{\mathcal{T}}\Delta & =_{df}\ \mathsf{exec}^{\dagger}(d[x])(\mathcal{T})\circ \mathsf{unfold}^{\dagger}(x)\Delta \\
[\![d]\!]_{\mathcal{T}}\Delta & =_{df}\ \mathsf{exec}^{\dagger}(d)(\mathcal{T})\Delta \\
[\![e_1; e_2]\!]_{\mathcal{T}}\Delta & =_{df}\ [\![e_2]\!]_{\mathcal{T}}\circ [\![e_1]\!]_{\mathcal{T}}\Delta \\
[\![x := e]\!]_{\mathcal{T}}\Delta & =_{df}\ [x'/x, r'/\mathtt{res}]([\![e]\!]_{\mathcal{T}}\Delta)\wedge x{=}r' \quad \textit{fresh logical } x', r' \\
[\![\mathtt{if}\ (v)\ e_1\ \mathtt{else}\ e_2]\!]_{\mathcal{T}}\Delta & =_{df}\ ([\![e_1]\!]_{\mathcal{T}}(v\wedge\Delta))\vee([\![e_2]\!]_{\mathcal{T}}(\neg v\wedge\Delta))
\end{array}$$

which form the foundation for us to analyse the loop body.

3.2 The Abstraction Mechanism

During symbolic execution, we may be confronted with infinitely many "concrete" program states and we need appropriate abstraction mechanisms to ensure the convergence of the fixed-point analysis. We design an abstraction function abs to abstract the (potentially infinite) concrete situations into finitely many abstract ones, aiming to obtain finiteness in the abstract domain. Our rationale of design is to keep only program variables and shared cutpoints; all other logical variables will be abstracted away. For example, $\mathsf{abs}(\mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{v_1},\mathtt{z_0}){*}\mathtt{llB}(\mathtt{z_0},\mathtt{n_1},\mathtt{S_1}))$ $= \mathtt{llB}(\mathtt{x},\mathtt{n},\mathtt{S})\wedge\mathtt{n}{=}\mathtt{n_1}{+}1\wedge\mathtt{S}{=}\mathtt{S_1}\sqcup\{\mathtt{v_1}\}$.

As illustrated, the abstraction transition function abs eliminates unimportant logic variable (during analysis) to ensure termination. Its type is defined as follows:

$$\mathsf{abs}\ :\ \mathsf{SH}\rightarrow\mathsf{SH} \qquad \text{Abstraction}$$

which indicates that it takes in a conjunctive abstract state σ and abstracts it as another conjunctive state σ'. Its rules are given below.

$$\mathsf{abs}(\sigma\wedge x_0{=}e) =_{df} \sigma[e/x_0] \qquad \mathsf{abs}(\sigma\wedge e{=}x_0) =_{df} \sigma[e/x_0]$$

$$\frac{x_0\notin \mathsf{Reach}(\sigma)}{\mathsf{abs}(\mathtt{H}(c)(x_0,v^*)*\sigma) =_{df} \sigma*\mathtt{true}}$$

$$\frac{p_2(u_2^*)\equiv\Phi \qquad \mathtt{H}(c_1)(x,v_1^*)*\sigma_1\vdash p_2(x,v_2^*)\wedge\pi_2 \qquad \mathsf{Reach}(p_2(x,v_2^*)*\sigma_3\wedge\pi_2)\cap\{v_1^*\}=\emptyset}{\mathsf{abs}(\mathtt{H}(c_1)(x,v_1^*)*\sigma_1*\sigma_3) =_{df} p_2(x,v_2^*)*\sigma_3\wedge\pi_2}$$

where $\mathtt{H}(c)(x,v^*)$ denotes $x{\mapsto}c(v^*)$ if c is a data node or $c(x,v^*)$ if c is a predicate. The function $\mathsf{Reach}(\sigma)$ returns all pointer variables which are reachable from program variables in the abstract state σ.

The first two rules eliminate logical variables (x_0) by replacing them with their equivalent expressions (e). The third rule is used to eliminate any garbage (heap part led by a logical variable x_0 unreachable from the other part of the heap) that may exist in the heap. The last rule of abs intends to eliminate shape formulae led by logical variables (all logical variables in v_1^*). It tries to fold some

data nodes $(\mathtt{H}(c_1)(x, v_1^*) * \sigma_1)$ up to one shape predicate $(p_2(x, v_2^*))$, and π_2 is the pure only residue. The predicate p_2 is selected from the user-defined predicates environments and it is the target shape to be abstracted against with. The rule ensures that the latter is a sound abstraction of the former by entailment proof, and the pointer logical parameters of c_1 are not reachable from the other part of the heap (so that the abstraction does not lose necessary information).

Then the lifting function is applied for abs to lift both its domain and range to disjunctive abstract states $\mathcal{P}_{\mathsf{SH}}$:

$$\mathsf{abs}^\dagger \bigvee \sigma_i =_{df} \bigvee \mathsf{abs}(\sigma_i)$$

3.3 Join and Widening

Join Operator. The operator join is applied over two conjunctive abstract states, trying to find a common shape as a sound abstraction for both:

$$\begin{array}{l}\mathsf{join}(\sigma_1, \sigma_2) =_{df} \\ \quad \textbf{let } \sigma_1', \sigma_2' = \mathsf{rename}(\sigma_1, \sigma_2) \textbf{ in} \\ \quad \textbf{match } \sigma_1', \sigma_2' \textbf{ with } (\exists x_1^* \cdot \kappa_1 \wedge \pi_1), (\exists x_2^* \cdot \kappa_2 \wedge \pi_2) \textbf{ in} \\ \quad\quad \textbf{if } \sigma_1 \vdash \sigma_2 * \mathtt{true} \textbf{ then } \exists x_1^*, x_2^* \cdot \kappa_2 \wedge (\mathsf{join}_\pi(\pi_1, \pi_2)) \\ \quad\quad \textbf{else if } \sigma_2 \vdash \sigma_1 * \mathtt{true} \textbf{ then } \exists x_1^*, x_2^* \cdot \kappa_1 \wedge (\mathsf{join}_\pi(\pi_1, \pi_2)) \\ \quad\quad \textbf{else } \sigma_1 \vee \sigma_2\end{array}$$

where the rename function avoids naming clashes among logical variables of σ_1 and σ_2, by renaming logical variables of same name in the two states with fresh names. For example, it will renew $\mathtt{x_0}$'s name in both states $\exists \mathtt{x_0} \cdot \mathtt{x_0}{=}\mathtt{0}$ and $\exists \mathtt{x_0} \cdot \mathtt{x_0}{=}\mathtt{1}$ to make them $\exists \mathtt{x_1} \cdot \mathtt{x_1}{=}\mathtt{0}$ and $\exists \mathtt{x_2} \cdot \mathtt{x_2}{=}\mathtt{1}$. After this procedure it judges whether σ_2 is an abstraction of σ_1, or the other way round. If either case holds, it regards the shape of the weaker state as the shape of the joined states, and performs joining for pure formulae with $\mathsf{join}_\pi(\pi_1, \pi_2)$, the join operator over pure domains [30,31]. Otherwise it keeps a disjunction of the two states (as it would be unsound to join their shapes together in this case).

For example, if we try

$$\mathsf{join}(\mathtt{x} \mapsto \mathtt{Node}(\mathtt{xmi_0}, \mathtt{xp_0}), \mathtt{slsB}(\mathtt{x}, \mathtt{xmi_0}, \mathtt{xmx_0}, \mathtt{xS_0}, \mathtt{xp_0}))\,,$$

since $\mathtt{x} \mapsto \mathtt{Node}(\mathtt{xmi_0}, \mathtt{xp_0})$ can be viewed as $\mathtt{slsB}(\mathtt{x}, \mathtt{xmi_0}, \mathtt{xmi_0}, \{\mathtt{xmi_0}\}, \mathtt{xp_0})$, the two arguments then can be joined as

$$\mathtt{slsB}(\mathtt{x}, \mathtt{xmi_0}, \mathtt{xmx_0}, \mathtt{xS_0}, \mathtt{xp_0}) \wedge \mathtt{xmi_0} \leq \mathtt{xmx_0} \wedge \{\mathtt{xmi_0}\} \sqsubseteq \mathtt{xS_0}.$$

We lift this operator for abstract state Δ as follows:

$$\mathsf{join}^\dagger(\Delta_1, \Delta_2) =_{df} \textbf{match } \Delta_1, \Delta_2 \textbf{ with } (\textstyle\bigvee_i \sigma_i^1), (\bigvee_j \sigma_j^2) \textbf{ in } \bigvee_{i,j} \mathsf{join}(\sigma_i^1, \sigma_j^2)$$

which essentially joins all pairs of disjunctions from the two abstract states, and makes a disjunction of them.

Widening Operator. The finiteness of the shape domain is confirmed by the abstraction function. To ensure the termination of the whole analysis, we still

need to guarantee the convergence of the analysis over the pure domain. This task is accomplished by the widening operator, which is defined as:

$$\begin{array}{l}\mathsf{widen}(\sigma_1,\sigma_2) =_{df}\\ \quad \textbf{let } \sigma_1',\sigma_2' = \mathsf{rename}(\sigma_1,\sigma_2) \textbf{ in}\\ \quad \textbf{match } \sigma_1',\ \sigma_2' \textbf{ with } (\exists x_1^*\cdot\kappa_1\wedge\pi_1),\ (\exists x_2^*\cdot\kappa_2\wedge\pi_2) \textbf{ in}\\ \quad\quad \textbf{if } \sigma_1\vdash\sigma_2 * \mathtt{true} \textbf{ then } \exists x_1^*,x_2^*\cdot\kappa_2\wedge(\mathsf{widen}_\pi(\pi_1,\pi_2))\\ \quad\quad \textbf{else } \sigma_1\vee\sigma_2\end{array}$$

In the widen operator, we expect that the second operand σ_2 to be weaker than the first σ_1, such that the widening reflects the trend of such weakening from σ_1 to σ_2. In this case, it returns the shape of σ_2 with pure part by applying widening operation $\mathsf{widen}_\pi(\pi_1,\pi_2)$ to the pure domain [30,31]. For example,

$$\begin{array}{l}\mathsf{widen}(\exists \mathtt{xmx_1},\mathtt{xS_1}\cdot\mathtt{slsB}(\mathtt{x},\mathtt{xmi_0},\mathtt{xmx_1},\mathtt{xS_1},\mathtt{xp_0})\wedge\mathtt{xv}{=}\mathtt{xmx_0}\wedge\mathtt{xS}{=}\mathtt{xS_1},\\ \qquad \exists \mathtt{xmx_2},\mathtt{xS_2}\cdot\mathtt{slsB}(\mathtt{x},\mathtt{xmi_0},\mathtt{xmx_2},\mathtt{xS_2},\mathtt{xp_0})\wedge\mathtt{xv}{\leq}\mathtt{xmx_2}\wedge\mathtt{xS}{\sqsubseteq}\mathtt{xS_2}){:=}\\ \exists \mathtt{xmx_0},\mathtt{xS_0}\cdot\mathtt{slsB}(\mathtt{x},\mathtt{xmi_0},\mathtt{xmx_0},\mathtt{xS_0},\mathtt{xp_0})\wedge\mathtt{xv}{\leq}\mathtt{xmx_0}\wedge\mathtt{xS}{\sqsubseteq}\mathtt{xS_0}.\end{array}$$

We also lift the operator over (disjunctive) abstract states:
$\mathsf{widen}^\dagger(\Delta_1,\Delta_2) =_{df} \textbf{match } \Delta_1,\Delta_2 \textbf{ with } (\bigvee_i \sigma_i^1),\ (\bigvee_j \sigma_j^2) \textbf{ in } \bigvee_{i,j}\mathsf{widen}(\sigma_i^1,\sigma_j^2)$

4 Specification Completion

From this section onwards, we present our invariant synthesis techniques for method pre/post-conditions. In this section, we focus on the scenario where pre/post shape templates are already supplied by users. Our proposed analysis will automatically refine such templates to complete specifications. We believe by allowing users to supply pre/post (shape) templates and letting the synthesis algorithm to compute the missing information (which is often tedious and error-prone), we offer a good way to harness the synergy between a human's insights and a machine's capability at automated program analysis.

The proposed specification synthesis algorithm (SPComplete) is given in Fig 4. The analysis proceeds in two steps for a method where the pre/post shape information is given in the specification, namely (1) forward analysis (at lines 1-2) and (2) pure constraint abstraction generation and solving (at lines 3-10).

The forward analysis Symb_Exec is shown in the right of Fig 4. Starting from a given pre-shape Φ_{pr}, it analyses the method body e (via symbolic execution; line 13) to compute the post-state in constraint abstraction form. The symbolic execution rules are similar to the rules shown in Section 3.1, except that a bi-abduction mechanism is used when the symbolic execution fails, and the failure location is recorded.

When symbolic execution fails to prove memory safety based on the current pre-state, the abduction mechanism is invoked. For example, if the current state is $\mathtt{llB}(\mathtt{x},\mathtt{n},\mathtt{S})$ (a list that is possibly empty), but $\mathtt{x}{\mapsto}\mathtt{node}(\_,\mathtt{p})$ is required by the next program instruction, our abduction mechanism will infer $\mathtt{n}{\geq}\mathtt{1}$ to add to the current state to satisfy the requirement.The variable *errLbls* (initialised at line 11) is to record the program locations in which abductions had occurred previously. Whenever the symbolic execution fails, it returns a state Δ that

Algorithm SPComplete($\mathcal{T}, mn, e, \Phi_{pr}, \Phi_{po}, u^*, v^*$)

1 Δ := Symb_Exec($\mathcal{T}, mn, e, \Phi_{pr}$)
2 **if** Δ = fail **then return** fail **end if**
3 Normalise Δ to DNF, and denote as $\bigvee_{i=1}^{m} \Delta_i$
4 w^*:=$\{u^*, v^*, v'^*\} \cup$ pureV($\{u^*, v^*, v'^*\}, \Phi_{pr} \vee \Phi_{po}$)
5 $\Delta_{\mathtt{P}}$:= Pure_CA_Gen($\Phi_{po}, \mathtt{Q}(w^*) ::= \bigvee_{i=1}^{m} \Delta_i$)
6 **if** $\Delta_{\mathtt{P}}$ = fail **then return** fail **end if**
7 π := Pure_CA_Solve($\mathtt{P}(w^*) ::= \Delta_{\mathtt{P}}$)
8 $R := t\ mn\ ((t\ u)^*; (t\ v)^*)$ *requires* ex_quan(Φ_{pr}, π) *ensures* ex_quan(Φ_{po}, π)
9 **if** Verify($\mathcal{T}, mn, R$) **then return** $\mathcal{T} \cup \{R\} \setminus \{ t\ mn((t\ u)^*; (t\ v)^*)\ requires\ \Phi_{pr}\ ensures\ \Phi_{po} \}$
10 **else return** fail **end if**

end Algorithm

Algorithm Symb_Exec ($\mathcal{T}, mn, e, \Phi_{pr}$)

11 *errLbls* := $\emptyset$
12 **do**
13 $(\Delta, l) := [\![e]\!]_{\mathcal{T}} mn(\Phi_{pr}, 0)$
14 **if** $l{>}0 \wedge l \notin$ *errLbls* **then**
15 Φ_{pr}:=ex_quan(Φ_{pr}, Δ);
16 *errLbls* := *errLbls* $\cup \{l\}$
17 **else if** $l{>}0 \wedge l \in$ *errLbls* **then return** fail
18 **end if**
19 **while** $l > 0$
20 **return** Δ

end Algorithm

Fig. 4. Specification Completion Algorithm

contains the abduction result and the location l where failure was detected, as shown in line 13. If the current abduction location l is not recorded in *errLbls*, it indicates that this is a new failure. The abduction result is added to the precondition of the current method to obtain a stronger Φ_{pr}, before the algorithm enters the symbolic execution loop with variable *errLbls* updated to add in the new failure location l. This loop is repeated until symbolic execution succeeds with no memory error, or a previous failure point was re-encountered. The latter may indicate a program bug or a specification error, or may be due to the possible incompleteness of the underlying Sleek prover we use.

Back to the main algorithm SPComplete, the analysis next builds a heap-based constraint abstraction, named $\mathtt{Q}(w^*)$, for the post-state in line 3. This constraint abstraction is possibly recursive. We then make use of another algorithm in Fig 5, named Pure_CA_Gen, to extract a pure constraint abstraction, named $\mathtt{P}(w^*)$, without any heap property. This algorithm tries to derive a branch $\mathtt{P}_i$ for each branch Δ_i of $\mathtt{Q}$. For every Δ_i it proceeds in two steps. In the first step (lines 2-4), it replaces the recursive occurrence of $\mathtt{Q}$ in Δ_i with $\sigma * \mathtt{P}(w^*)$. In the second step (lines 5-6) it tries to derive $\mathtt{P}_i$ via the entailment. If the entailment fails, then abduction is used to discover any missing constraint σ'_i for $\rho\Delta_i$ to allow the entailment to succeed. In this case, σ'_i is incorporated into σ_i (and eventually $\mathtt{P}_i$). Once this is done, we use some existing fixpoint analysis (e.g. [31]) inside SPComplete to derive non-recursive constraint π, as a simplification of $\mathtt{P}(w^*)$. This result is then incorporated into the pre/post specifications in line 8, before we perform a post verification in line 9 using the Hip verifier [29], to ensure the strengthened precondition is strong enough for memory safety.

The function pureV(V, Δ) retrieves from Δ the shapes referred to by all pointer variables from V, and returns the set of logical variables used to record numerical (size and bag) properties in these shapes, e.g. pureV($\{\mathtt{x}\}, \mathtt{ll}(\mathtt{x}, \mathtt{n})$) returns

Algorithm Pure_CA_Gen($\sigma, \mathtt{Q}(w^*) ::= \bigvee_{i=1}^{m} \Delta_i$)

1 **for** $i = 1$ **to** m
2 Denote all appearances of $\mathtt{Q}(w^*)$ in Δ_i as $\mathtt{Q}_j(w_j^*), j = 1, ..., p$
3 Denote substitutions $\rho_j = [([w_j^*/w^*]\sigma * \mathtt{P}(w_j^*))/\mathtt{Q}_j(w_j^*)]$
4 Let substitution $\rho := \rho_1 \circ \rho_2 \circ ... \circ \rho_p$ as applying all substitutions defined above in sequence
5 **if** $(\rho\Delta_i \vdash \sigma * \sigma_i$ **or** $\rho\Delta_i \wedge [\sigma_i'] \rhd \sigma * \sigma_i)$ **and** $ispure(\sigma_i)$ **then** $\mathtt{P}_i := \sigma_i$
6 **else return** fail **end if**
7 **end for**
8 **return** $\bigvee_{i=1}^{m} \mathtt{P}_i$

end Algorithm

Fig. 5. Pure constraint abstraction generation algorithm

$\{\mathtt{n}\}$. This function is used in the algorithm to ensure that all free variables in Φ_{pr} and Φ_{po} are added into the parameter list of the constraint abstraction $\mathtt{Q}$. The function $\mathsf{ex\_quan}(\Delta, \pi)$ is to strengthen the state Δ with the abduction result π: $\mathsf{ex\_quan}(\Delta, \pi) =_{df} \Delta \wedge \exists(\mathsf{fv}(\pi) \setminus \mathsf{fv}(\Delta)) \cdot \pi$. It is used to incorporate the discovered missing pure constraints into the original specification. For example, $\mathsf{ex\_quan}(\mathtt{ll(x,n)}, \mathtt{0<m \wedge m \leq n})$ returns $\mathtt{ll(x,n)} \wedge \mathtt{0<n}$. The bi-abduction mechanism is defined in Section 4.1.

4.1 The Bi-abduction

We present a bi-abduction procedure over our combined domain as a generalisation to the shape bi-abduction [8] which caters for only the shape domain.

The bi-abduction procedure $\sigma * [\sigma_m] \rhd \sigma_1 * \sigma_f$ aims to find the anti-frame part σ_m (*the missing part*) and the frame part σ_f such that $\sigma * \sigma_m \vdash \sigma_1 * \sigma_f$ with given σ and σ_1. Our abduction procedure can handle more than one predicates in the analysis, while the shape abduction [8] caters for only one specified shape predicate domain. Another advance is that we can infer numerical and bag properties together with the shape formulae for the anti-frame part to improve the precision of the analysis. The bi-abduction procedure is calculated by $\mathtt{BiAbd}(\sigma, \sigma_1) = (\sigma_m, \sigma_f)$ function (shown in Fig. 6).

The $\mathtt{Entail}(\sigma, \sigma_1)$ function returns σ_f if the SLEEK entailment $\sigma \vdash \sigma_1 * \sigma_f$ succeeds with σ_f, or `false` if the entailment $\sigma \vdash \sigma_1 * \mathtt{true}$ fails.

The $\mathtt{BiAbd}(\sigma, \sigma_1)$ function checks whether $\sigma \vdash \sigma_1 * \sigma_1'$ for some σ_1' first. If the entailment proof succeeds, we do not need abduction, and return `emp` as the anti-frame and σ_1' as the frame.

The function $\mathtt{BiAbd}_1$ triggers when the LHS (σ) does not entail the RHS (σ_1) but the RHS entails the LHS with some formula (σ_1') as the residue. This function is quite general and applies in many cases. For instance, for the formula $\mathtt{emp} \nvdash \mathtt{x} \mapsto \mathtt{Node(xv,xp)}$, the RHS can entail the LHS with residue $\mathtt{x} \mapsto \mathtt{Node(xv,xp)}$. The abduction then checks whether σ plus the frame information σ_1' implies $\sigma_1 * \sigma_2'$

$\texttt{BiAbd}(\sigma, \sigma_1) =_{df}$
  let $\sigma_1' = \texttt{Entail}(\sigma, \sigma_1)$ **in**
  if $\sigma_1' \neq \texttt{false}$ **then** $(\texttt{emp}, \sigma_1')$ **else** $\texttt{BiAbd}_1(\sigma, \sigma_1)$
$\texttt{BiAbd}_1(\sigma, \sigma_1) =_{df}$
  let $\sigma_1' = \texttt{Entail}(\sigma_1, \sigma)$ **and** $\sigma_2' = \texttt{Entail}(\sigma * \sigma_1', \sigma_1)$ **in**
  if $\sigma_1' \neq \texttt{false} \wedge \sigma_2' \neq \texttt{false}$ **then** (σ_1', σ_2') **else** $\texttt{BiAbd}_2(\sigma, \sigma_1)$
$\texttt{BiAbd}_2(\sigma, \sigma_1) =_{df}$
  let $\sigma_u' \in \mathsf{unroll}(\sigma)$ **and** $(\sigma_0', \sigma_1') = \texttt{BiAbd}(\sigma_u', \sigma_1)$ **and** $\sigma_2' = \texttt{Entail}(\sigma * \sigma_1', \sigma_1)$ **in**
  if $\mathsf{data\_no}(\sigma_u') \leq \mathsf{data\_no}(\sigma_1) \wedge \sigma_1' \neq \texttt{false} \wedge \sigma_2' \neq \texttt{false}$
  then (σ_1', σ_2') **else** $\texttt{BiAbd}_3(\sigma, \sigma_1)$
$\texttt{BiAbd}_3(\sigma, \sigma_1) =_{df}$
  if the caller is not $\texttt{BiAbd}_3$ **then**
    let $(\sigma_0', \sigma_1') = \texttt{BiAbd}(\sigma_1, \sigma)$ **and** $\sigma_2' = \texttt{Entail}(\sigma * \sigma_1', \sigma_1)$ **in**
    if $\sigma_1' \neq \texttt{false} \wedge \sigma_2' \neq \texttt{false}$ **then** (σ_1', σ_2') **else** $\texttt{BiAbd}_4(\sigma, \sigma_1)$
  else $\texttt{BiAbd}_4(\sigma, \sigma_1)$
$\texttt{BiAbd}_4(\sigma, \sigma_1) =_{df}$
  let $\sigma_2' = \texttt{Entail}(\sigma * \sigma_1, \sigma_1)$ **in**
  if $\sigma_2' \neq \texttt{false}$ **then** (σ_1, σ_2') **else** $(\texttt{false}, \texttt{false})$

Fig. 6. Bi-Abduction Algorithm

for some σ_2' (emp in this example), and returns $\texttt{x}{\mapsto}\texttt{Node}(\texttt{xv}, \texttt{xp})$ as the anti-frame. If the function is not applicable, we use the function $\texttt{BiAbd}_2$.

The function $\texttt{BiAbd}_2$ deals with the case where neither side entails the other, e.g. for $\texttt{slsB}(\texttt{x}, \texttt{xmi}, \texttt{xmx}, \texttt{xS}, \texttt{null})$ as σ and $\exists \texttt{p}, \texttt{u}, \texttt{v} \cdot \texttt{x}{\mapsto}\texttt{Node}(\texttt{u}, \texttt{p}) * \texttt{p}{\mapsto}\texttt{Node}(\texttt{v}, \texttt{null})$ as σ_1. As the shape predicates in the antecedent σ are formed by disjunctions according to their definitions (like slsB), its certain disjunctive branches may imply σ_1. In this function, we first unfold σ ($\sigma_u' \in \mathsf{unroll}(\sigma)$) and try further abduction with the results (σ_u') against σ_1. If it succeeds with a frame σ_1', then we confirm the abduction by ensuring that $\texttt{Entail}(\sigma * \sigma_1', \sigma_1)$ succeeds. For the example above, the abduction returns $\exists \texttt{u}, \texttt{v} \cdot \texttt{xS} = \{\texttt{u}, \texttt{v}\}$ as the anti-frame σ_1' and discovers the nontrivial frame $\texttt{u}{=}\texttt{xmi} \wedge \texttt{v}{=}\texttt{xmx} \wedge \texttt{u} \leq \texttt{v}$ as σ_2'. The function data_no returns the number of data nodes in a state, e.g. it returns 1 for $\texttt{x}{\mapsto}\texttt{Node}(\texttt{v}, \texttt{p}) * \texttt{llB}(\texttt{p}, \texttt{n}, \texttt{T})$. This syntactic check prevents unlimited number of times of unrolling from happening when the abduction procedure invokes this function recursively. The unroll unfolds all shape predicates once in σ, normalises the result to a disjunctive form ($\bigvee_{i=1}^{n} \sigma_i$), and returns the result as a set of formulae ($\{\sigma_1, ..., \sigma_n\}$).

The function $\texttt{BiAbd}_3$ is applied when $\texttt{BiAbd}_2$ does not work, e.g. $\exists \texttt{p}, \texttt{u}, \texttt{v}, \texttt{q} \cdot \texttt{x}{\mapsto}\texttt{Node}(\texttt{u}, \texttt{p}) * \texttt{p}{\mapsto}\texttt{Node}(\texttt{v}, \texttt{q})$ as σ and $\exists \texttt{xS} \cdot \texttt{slsB}(\texttt{x}, \texttt{xmi}, \texttt{xmx}, \texttt{xS}, \texttt{xp})$ as σ_1. In this case the antecedent σ cannot be unfolded as it contains only data nodes. As the function defines, it reverses two sides of the entailment and applies the BiAbd to uncover the constraints σ_0' and σ_1'. Then it checks that whether σ with σ_1' conjoined, does entail σ_1 before it returns σ_2'. For the example above, the anti-frame (σ_1') is inferred as $\texttt{u} \leq \texttt{v}$. The caller of this function is not allowed to be $\texttt{BiAbd}_3$, in order to prevent infinite number of applications of this function.

If the first four functions do not succeed in finding a solution, the last function $\texttt{BiAbd}_4$ is invoked to add the consequence σ_1 to the antecedent σ, provided that

they are consistent. It is effective for situations like $\texttt{x}{\mapsto}\texttt{Node}(\_,\_) \not\vdash \texttt{y}{\mapsto}\texttt{Node}(\_,\_)$, where we should add $\texttt{y}{\mapsto}\texttt{node}(\_,\_)$ to the LHS directly (as the other four functions do not apply here). In our analysis, we assume that different variables refer to different nodes unless aliasing is suggested in the program code. For example, the if-statement $\texttt{if}\ (\texttt{x} == \texttt{y})\{\texttt{c}\}$ suggests that x and y are aliased in code c.

5 Postcondition Synthesis

In this section, we work on the scenario where the precondition for a method is given but not the postcondition. We propose an invariant synthesis algorithm to compute the missing postcondition starting from a given precondition. As shown in Fig. 7, the algorithm has three inputs: $\mathcal{T}$ is the set of methods with their specifications, the procedure to be analysed with its precondition Pre, and a pre-set upper bound n (the maximal number of cutpoints to keep track of).

PostInfer($\mathcal{T}$, $t\ mn\ ((t\ x)^*;(t\ y)^*)$ *requires* Pre $\{e\}$, n)
Local: $i := 0$; $\mathsf{Post}_i :=$ **false**;
1 $\quad\mathcal{T}' := \mathcal{T} \cup \{t\ mn\ ((t\ x)^*;(t\ y)^*)$ *requires* Pre *ensures* Post_i $\{e\}\}$;
2 $\quad$**repeat**
3 $\quad\quad i := i + 1$;
4 $\quad\quad \mathsf{Post}_i := \mathsf{abs}^\dagger([\![e]\!]_{\mathcal{T}'}(\mathsf{Pre}))$;
5 $\quad\quad \mathsf{Post}_i := \mathsf{widen}^\dagger(\mathsf{Post}_{i-1}, \mathsf{join}^\dagger(\mathsf{Post}_{i-1}, \mathsf{Post}_i))$;
6 $\quad\quad$**if** Post_i=false **or** cp_no(Post_i)$>n$ **then return** fail **end if**
7 $\quad\quad \mathcal{T}' := \mathcal{T} \cup \{t\ mn\ ((t\ x)^*;(t\ y)^*)$ *requires* Pre *ensures* Post_i $\{e\}\}$;
8 $\quad$**until** $\mathsf{Post}_i = \mathsf{Post}_{i-1}$
9 $\quad$**return** Post_i

Fig. 7. Postcondition synthesis algorithm

As we see before, the algorithm is similar to the loop invariant synthesis algorithm. One difference is we set the initial postcondition of the method t to be the strongest condition false, which will be used as the initial environment for recursive method calls. In the following process, the postcondition is weakened until a fixed-point is achieved. For each iteration, we symbolically execute the method body by using the current specification, and generate a postcondition for the current method body (line 4). After applying the abstraction, join and widening operations to the current condition, a new postcondition is updated as the method specification (line 5-7), and will be used in the next iteration. After each iteration, we test whether a fixpoint is reached (line 8), if so, we return the current postcondition as the finial result. During the analysis, if the number of logical variables maintained oversteps the specified bound n, the algorithm is stopped and a failure is reported (line 6).

The operations used in the algorithm, such as the abstract semantics $[\![e]\!]_{\mathcal{T}}\Delta$, the abstraction function $\mathsf{abs}^\dagger$, and $\mathsf{join}^\dagger$ and $\mathsf{widen}^\dagger$ are defined in Section 3.

6 Precondition Synthesis

Writing preconditions is also a cumbersome and error-prone task. In this section, we discuss an approach to synthesise preconditions in the combined domain. Our proposed analysis algorithm is given in Fig. 8. It takes three input parameters: $\mathcal{T}$ as the set of method specifications that are already inferred, the procedure to be analysed $t\ mn\ ((t\ x)^*;(t\ y)^*)\ \{e\}$, and a pre-set upper bound n on the number of shared logical variables that we keep during the analysis.

To infer the preconditions in the combined shape and pure domain, we apply the bi-abduction technique in the abstract interpretation framework over the combined domain. During the analysis, we keep track of a pair of formulae (Δ_1, Δ_2) representing the inferred precondition and the current state respectively. As in a standard abstract interpretation framework, our analysis carries out the fixed-point iteration until a fixed-point is reached.

PreInfer($\mathcal{T}$, $t\ mn\ ((t\ x)^*;(t\ y)^*)\ \{e\}$, n)
Local: $i := 0$; $\mathsf{Pre}_i := \mathbf{emp}, \mathsf{Post}_i := \mathbf{false}$;
1 $\mathcal{T}' := \mathcal{T} \cup \{t\ mn\ ((t\ x)^*;(t\ y)^*)\ \mathit{requires}\ \mathsf{Pre}_0\ \mathit{ensures}\ \mathsf{Post}_0\ \{e\}\}$;
2 **repeat**
3 $i := i + 1$;
4 $(\mathsf{Pre}_i, \mathsf{Post}_i) := [\![e]\!]^{\mathsf{A}}_{\mathcal{T}'}(\mathsf{Pre}_{i-1}, \mathsf{Pre}_{i-1})$;
5 $(\mathsf{Pre}_i, \mathsf{Post}_i) := (\mathsf{abs}^{\dagger}(\mathsf{Pre}_i), \mathsf{abs}^{\dagger}(\mathsf{Post}_i))$;
6 $(\mathsf{Pre}_i, \mathsf{Post}_i) := (\mathsf{join}^{\dagger}(\mathsf{Pre}_{i-1}, \mathsf{Pre}_i), \mathsf{join}^{\dagger}(\mathsf{Post}_{i-1}, \mathsf{Post}_i))$;
7 $(\mathsf{Pre}_i, \mathsf{Post}_i) := (\mathsf{widen}^{\dagger}(\mathsf{Pre}_{i-1}, \mathsf{Pre}_i), \mathsf{widen}^{\dagger}(\mathsf{Post}_{i-1}, \mathsf{Post}_i))$;
8 **if** Pre_i=`false` **or** Post_i=`false` **or** $\mathsf{cp\_no}(\mathsf{Pre}_i) > n$ **or** $\mathsf{cp\_no}(\mathsf{Post}_i) > n$ **then return** fail **end if**
9 $\mathcal{T}' := \mathcal{T} \cup \{t\ mn\ ((t\ x)^*;(t\ y)^*)\ \mathit{requires}\ \mathsf{Pre}_i\ \mathit{ensures}\ \mathsf{Post}_i\ \{e\}\}$;
10 **until** $\mathsf{Pre}_i = \mathsf{Pre}_{i-1}$ does not change
11 $\mathsf{Post} = [\![e]\!]_{\mathcal{T}'}\mathsf{Pre}_i$;
12 **if** Post = `false` **then return** fail **else return** Pre_i
13 **end if**

Fig. 8. Precondition synthesis algorithm

We first set the method precondition as `emp` and postcondition as `false` which signifies that we know nothing about the method (line 1). Then for each iteration, a forward abductive analysis is employed to compute a new pre-/post-condition (line 4) based on the current specification. The analysis performs abstraction on both obtained pre-/post-conditions to maintain the finiteness of the shape domain. The obtained results are joined with the results from the previous iteration (line 6), and a widening is conducted over both to ensure termination of the analysis (line 7). If the analysis cannot continue due to a program bug, or cannot keep the number of shared logical variables/cutpoints (counted by $\mathsf{cp\_no}$) within a specified bound (n), then a failure is reported (line 8). At the end of each iteration, the inferred specification is used to update the specification of mn

(line 9), which will be used for recursive calls (if any) of mn in next iteration. Finally we judge whether a fixed-point is already reached (line 10). The last few lines (from line 11) ensure that inferred precondition are indeed sound using a standard abstract semantics (without abduction). Any unsound specifications will be ruled out.

6.1 Abstract Semantics with Abduction

As shown in the algorithm, we use two kinds of abstract semantics to analyse the program: an abstract semantics with abduction to derive the specification for the program (line 4), and another underlying semantics to ensure soundness for the analysis result (line 11) which is defined in Section 3.1.

We shall now define the abstract semantics with abduction, which is of the form

$$[\![e]\!]^{\mathsf{A}} \ : \ \mathsf{AllSpec} \rightarrow \mathcal{P}(\mathsf{SH} \times \mathsf{SH}) \rightarrow \mathcal{P}(\mathsf{SH} \times \mathsf{SH})$$

It takes a piece of program and a specification table, and maps a (disjunctive) set of pairs of symbolic heaps to another such set (where the first in the pair is the accumulated precondition and the second is the current state).

This semantics also consists of the basic transition functions which compose the atomic instructions' semantics and then the program constructors' semantics. Here the basic transition functions are lifted as

$$\begin{array}{l}
\mathsf{Unfold}(x)(\sigma', \sigma) =_{df} \\
\quad \textbf{let } \Delta{=}\mathsf{unfold}(x)\sigma \text{ and } \mathsf{S}{=}\{(\sigma', \sigma_1) \mid \sigma_1 \in \Delta\} \\
\quad \textbf{in if } (\mathtt{false} \notin \Delta) \textbf{ then } \mathsf{S} \\
\quad\quad \textbf{else if } (\Delta \vdash x{=}a \text{ for some } a{\in}\mathsf{SVar}) \text{ and} \\
\quad\quad\quad\quad (\sigma' \not\vdash c(a, y^*) * \mathtt{true} \text{ for fresh } \{y^*\}{\subseteq}\mathsf{LVar}) \\
\quad\quad \textbf{then } \mathsf{S} \cup \{(\sigma' * c(x, y^*), \Delta * c(x, y^*))\} \\
\quad\quad \textbf{else } \ \mathsf{S} \cup \{(\sigma', \mathtt{false})\}
\end{array}$$

$$\begin{array}{l}
\mathsf{Exec}(ds)(\sigma', \sigma) =_{df} \textbf{let } \sigma_1{=}\mathsf{exec}(ds)\sigma \textbf{ in } \{(\sigma', \sigma_1) \mid \sigma_1 \in \Delta\} \\
\quad\quad \text{where } ds \text{ is either } d[x] \text{ or } d, \text{ excluding procedure call}
\end{array}$$

$$\frac{\begin{array}{c} t\ mn\ ((t_i\ u_i)_{i=1}^m; (t_i\ v_i)_{i=1}^n)\ \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} \in \mathcal{T} \\ \rho = [x_i'/u_i]_{i=1}^m \circ [y_i'/v_i]_{i=1}^n \qquad \sigma * [\sigma_1'] \rhd \rho\Phi_{pr} * \sigma_1 \\ \rho_o = [r_i/v_i]_{i=1}^n \circ [x_i'/u_i']_{i=1}^m \circ [y_i'/v_i']_{i=1}^n \quad \rho_l = [r_i/y_i']_{i=1}^n \quad \mathit{fresh\ logical}\ r_i \end{array}}{\begin{array}{l} \mathsf{Exec}(mn(x_1, .., x_m; y_1, .., y_n))(\mathcal{T})(\sigma, \sigma') =_{df} \\ \qquad \{(\sigma_2, \rho_o(\sigma' * \sigma_1')) \mid \sigma_2 \in (\rho_l\sigma_1) * (\rho_o\Phi_{po})\} \end{array}}$$

A similar lifting function † is defined to lift Unfold's and Exec's domains:

$$\begin{array}{ll}
\mathsf{Unfold}^{\dagger}(x) \bigvee(\sigma_i', \sigma_i) & =_{df} \bigvee(\mathsf{Unfold}(x)(\sigma_i', \sigma_i)) \\
\mathsf{Exec}^{\dagger}(ds)(\mathcal{T}) \bigvee(\sigma_i', \sigma_i) & =_{df} \bigvee(\mathsf{Exec}(ds)(\mathcal{T})(\sigma_i', \sigma_i))
\end{array}$$

Based on the above transition functions, the abstract semantics with abduction is as follows:

$$
\begin{aligned}
[\![d[x]]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', \Delta) &=_{df} \mathsf{Exec}^{\dagger}(d[x])(\mathcal{T}) \circ \mathsf{Unfold}^{\dagger}(x)(\Delta', \Delta) \\
[\![d]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', \Delta) &=_{df} \mathsf{Exec}^{\dagger}(d)(\mathcal{T})(\Delta', \Delta) \\
[\![e_1; e_2]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', \Delta) &=_{df} [\![e_2]\!]^{\mathsf{A}}_{\mathcal{T}} \circ [\![e_1]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', \Delta) \\
[\![x := e]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', \Delta) &=_{df} [x'/x, r'/\mathtt{res}]([\![e]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', \Delta)) \wedge x{=}r' \\
&\qquad\qquad \textit{fresh logical } x', r' \\
[\![\mathtt{if}\ (v)\ e_1\ \mathtt{else}\ e_2]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', \Delta) &=_{df} ([\![e_1]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', v{\wedge}\Delta)) \vee ([\![e_2]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', \neg v{\wedge}\Delta))
\end{aligned}
$$

For assignment we apply the substitution on both abstract states in the pair.

7 Experiments and Evaluation

We have implemented a prototype system and evaluated it over a number of heap-manipulating programs to test the viability and precision of our approaches in different scenarios. Our experimental results were achieved with an Intel Core i7 CPU 2.66GHz with 8GB RAM. We have also defined a library of predicates covering popular data structures and a variety of properties. The predicates required as input by our tool can be selected from the library or can be supplied by users, according to the input program data structures and the properties of interest. Usually, the upper bound of cutpoints is set to be twice the number of input program variables to improve the precision. Some of our results are presented in Table 1. The tables shows the analysed methods and the analysis time (in seconds) for different scenarios, such as loop invariant synthesis (**LoopInv**), pre/post-condition completion (**SPComplt**), postcondition inference (**PostInfer**) and precondition inference (**PreInfer**). Some methods in the table, like `height` that calculates the height of the input tree, and `count` that returns the number of nodes in the input tree are not tail recursive function, and such methods cannot be written as loops. We leave the time slots empty.

Comparing our approach to the previous approaches the first observation concerns the precision of our analysis. Our tool uses a combined domain, it can discover more expressive specifications to guarantee memory safety and functional correctness. For example in case of the `take` program which traverses the list down for a user-specified number `n` of nodes, we can find that the input list length must be no less than `n`. However the previous tools based on a shape domain (like [9]) can only discover the memory safety properties of given programs. Since our shape domain includes tree data structures, our tool is able to discover complex functional specifications for binary search trees in contrast to the previous approaches [10,27]. For example in case of the `flatten` method, our tool is able to discover that the input data structure is a binary search tree while the output data structure is a sorted doubly linked list having the same data content (values stored inside the nodes) as that of the input.

The second observation regarding our experimental results is that the specification completion algorithm uses less time than other three algorithms in general. This is because the **SPComplt** algorithm does not need fixed-point computation in the combined domain. The time usage of loop invariant synthesis and

Table 1. Selected Experimental Results

Prog.	**LoopInv**	**SPComplt**	**PostInfer**	**PreInfer**
Singly Linked List				
create	0.45	0.22	0.47	0.82
delete	0.72	0.42	0.78	1.12
traverse	0.63	0.29	0.67	0.92
length	0.77	0.42	0.75	0.86
append	0.31	0.44	0.43	1.02
take	0.85	0.52	0.81	0.86
insert	0.84	0.57	0.88	0.84
reverse	1.03	0.56	1.07	1.16
filter	1.18	0.72	1.12	1.56
Sorting algorithm				
sort_insert	0.82	0.47	0.84	1.92
merge	1.97	1.12	1.83	2.78
sort_select	0.69	0.47	0.67	1.06
Doubly Linked List				
create	0.56	0.33	0.53	0.94
append	0.76	0.47	0.72	1.72
insert	0.68	0.42	0.62	1.42
Binary Search Tree				
create	-	0.71	1.42	1.68
insert	1.36	0.87	1.32	1.41
search	1.29	0.98	1.22	1.72
height	-	0.62	0.98	1.03
count	-	0.77	0.94	1.05
flatten	-	1.09	1.72	2.92

postcondition inference are almost the same, it is because the both algorithms have the same computation strategy. The precondition inference algorithm need most time over all the others, since it requires both bi-abduction and fixed-point computation in the combine domain. From the comparison, we can conclude that, if users want the verification to be faster, more information need to be given. If the level of automation of verification is the first concern, then the verification system will need more time to check and compute.

8 Related Work and Conclusion

Dramatic advances have been made in synthesising heap-manipulating programs' specifications. The local shape analysis [14] infers loop invariants for list-processing programs, followed by the SpaceInvader/Abductor tool to infer full method specifications over the separation domain, so as to verify pointer safety for larger industrial codes [9,42]. The SLAyer tool [16] implements an inter-procedural analysis for programs with shape information. Compared with them, our abstraction is more general since it is driven by predicates and is

not restricted to linked lists. To deal with size information (such as number of nodes in lists/trees), THOR [28] derives a numerical program from the original heap-processing one, such that the size properties can be obtained by numerical analysis. A similar approach [17] combines a set domain (for shape) with its cardinality domain (for corresponding numerical information) in a more general framework. Compared with these works, our approach can discover specifications with stronger invariants such as sortedness and bag-related properties, which have not been addressed in the previous works. One more work to be mentioned is the relational inductive shape analysis [10,25] and our previous inference works [33,34]. These works can handle shape and numerical information over a combined domain. However they still require user given preconditions for the program and only deal with specific scenarios, whereas we can also compute the preconditions. Rival and Chang [38] propose an inductive predicate to summarize call stacks along with heap structures in a context of a whole-program analysis. In contrast our analysis is modular, inferring an abstraction of a procedure heap effect.

There are also other approaches that can synthesise shape-related program invariants other than those based on separation logic. The shape analysis framework TVLA [41] is based on three-valued logic. It is capable of handling complicated data structures and properties, such as sortedness. Guo et al. [18] report a global shape analysis that discovers inductive structural shape invariants from the code. Kuncak et al. [23] develop a role system to express and track referencing relationships among objects, where an object's role (type) depends on, and changes according to, the mutation of its referencing. Hackett and Rugina [19] can deal with AVL-trees but it is customised to handle only tree-like structures with height property. Bouajjani et al. [5,6] propose a program analysis in an abstract domain with SL3 (Singly-Linked List Logic) and size, sortedness and multi-set properties. However, their heap domain is restricted to singly-linked list only, and their shape analysis is separated from numerical and mutli-set analyses. Type-based approaches [39,40] are also used to infer numerical constraints for given type templates, but limited to capture flow sensitive constraints. Compared with these works, separation logic based approaches benefit from the frame rule with support for local reasoning.

There are other approaches which unify reasoning over shape and data using either a combination of appropriate decision procedures inside Satisfiability-Modulo-Theories (SMT) solvers (e.g. [36,24]) or a combination of appropriate abstract interpreters inside a software model checker (e.g. [4]). Compared with our work, their heap domains are mainly restricted to linked lists.

Concluding Remarks. We have reported in this paper program analysis techniques that help synthesise automatically program invariants over a combined separation and pure(numerical+bag) domain. Our inference mechanisms include a loop invariant synthesis, a static analysis that refines (partial) pre/post shape templates to (complete) specifications, a postcondition inference and a precondition synthesis. The key components of the proposed analyses include novel

abstraction and abduction mechanisms, join and widening operators. We have built a prototype system and the initial experimental results are encouraging.

Acknowledgement. This work was supported by EPSRC project EP/G042322.

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Software and System Modeling 4 (2005)
2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
4. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)
5. Bouajjani, A., Dragoi, C., Enea, C., Sighireanu, M.: On inter-procedural analysis of programs with lists and data. In: PLDI (2011)
6. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: Abstract domains for automated reasoning about list-manipulating programs with infinite data. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 1–22. Springer, Heidelberg (2012)
7. Bozga, M., Iosif, R., Lakhnech, Y.: Storeless semantics and alias logic. In: PEPM (2003)
8. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL (2009)
9. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM 58(6) (2011)
10. Chang, B.Y.E., Rival, X.: Relational inductive shape analysis. In: POPL (2008)
11. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. of Comp. Prog. 77 (2012)
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1977)
13. Deutsch, A.: Interprocedural may-alias analysis for pointers: Beyond -limiting. In: PLDI (1994)
14. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
15. Filliâtre, J.-C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013)

16. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 240–260. Springer, Heidelberg (2006)
17. Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: Shao, Z., Pierce, B.C. (eds.) POPL (2009)
18. Guo, B., Vachharajani, N., August, D.I.: Shape analysis with inductive recursion synthesis. In: PLDI (2007)
19. Hackett, B., Rugina, R.: Region-based shape analysis with tracked locations. In: POPL (2005)
20. Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: POPL (2001)
21. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: Verifast: A powerful, sound, predictable, fast verifier for c and java. In: NASA Formal Methods (2011)
22. Jonkers, H.: Abstract storage structures. Algorithmic Languages (1981)
23. Kuncak, V., Lam, P., Rinard, M.C.: Role analysis. In: POPL (2002)
24. Lahiri, S.K., Qadeer, S.: Back to the future: revisiting precise program verification using smt solvers. In: POPL (2008)
25. Laviron, V., Chang, B.-Y.E., Rival, X.: Separating shape graphs. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 387–406. Springer, Heidelberg (2010)
26. Leino, K.R.M., Müller, P., Smans, J.: Verification of concurrent programs with Chalice. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD 2007/2008/2009. LNCS, vol. 5705, pp. 195–222. Springer, Heidelberg (2009)
27. Magill, S., Tsai, M.-H., Lee, P., Tsay, Y.-K.: THOR: A tool for reasoning about shape and arithmetic. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 428–432. Springer, Heidelberg (2008)
28. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Automatic numeric abstractions for heap-manipulating programs. In: POPL (2010)
29. Nguyen, H.H., David, C., Qin, S., Chin, W.-N.: Automated verification of shape and size properties via separation logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)
30. Pham, T.-H., Trinh, M.-T., Truong, A.-H., Chin, W.-N.: FixBag: A fixpoint calculator for quantified bag constraints. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 656–662. Springer, Heidelberg (2011)
31. Popeea, C., Chin, W.-N.: Inferring disjunctive postconditions. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 331–345. Springer, Heidelberg (2008)
32. Qin, S., He, G., Luo, C., Chin, W.-N.: Loop invariant synthesis in a combined domain. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 468–484. Springer, Heidelberg (2010)
33. Qin, S., He, G., Luo, C., Chin, W.N., Chen, X.: Loop invariant synthesis in a combined abstract domain. J. Symb. Comput. 50 (2013)
34. Qin, S., He, G., Luo, C., Chin, W.N., Yang, H.: Automatically refining partial specifications for heap-manipulating programs. Sci. Comput. Program. (accepted to appear)
35. Qin, S., Luo, C., Chin, W.-N., He, G.: Automatically refining partial specifications for program verification. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 369–385. Springer, Heidelberg (2011)

36. Rakamarić, Z., Bruttomesso, R., Hu, A.J., Cimatti, A.: Verifying heap-manipulating programs in an SMT framework. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 237–252. Springer, Heidelberg (2007)
37. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS (2002)
38. Rival, X., Chang, B.Y.E.: Calling context abstraction with shapes. In: POPL (2011)
39. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: PLDI (2008)
40. Rondon, P.M., Kawaguchi, M., Jhala, R.: Low-level liquid types. In: POPL (2010)
41. Sagiv, M., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3) (2002)
42. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)

Slow Abstraction via Priority

A.W. Roscoe and Philippa J. Hopcroft

Oxford University Department of Computer Science,
and Verum Software Technologies BV

Abstract. CSP treats internal τ actions as *urgent*, so that an infinite sequence of them is the misbehaviour known as *divergence*, and states with them available make no offer that we can rely on. While it has been possible to formulate a number of forms of abstraction in these models where the abstracted actions become τs, it has sometimes been necessary to be careful about the interpretation of τs and divergence. In this paper, inspired by an industrial problem, we demonstrate how this range of abstractions can be extended to encompass the offers made by processes during a run of "slow τs", namely abstractions of interactions with an external agent that does not usually respond urgently to an offer, but always eventually does respond. This extension requires the *prioritise* operator recently introduced into CSP and its refinement checker FDR. We demonstrate its use in the modelling used in Verum's ASD:Suite.

1 Introduction

Hoare's CSP [7, 12, 13] treats the actions a process can perform alike, except that while ordinary visible communications in the alphabet Σ require the agreement of the external environment to occur, the special action τ does not.

The CSP hiding operator $P \setminus X$ makes the assumption that the hidden X actions (now τs) happen as soon as they can. However, hiding is also a means of abstracting part of the external interface of a process, and this has meant that in formulating concepts such as $\mathcal{L}_A(P)$ (lazy abstraction [12]) we have had to allow for abstracted actions not being *urgent*. Lazy abstraction assumes that abstracted events may occur inside $\mathcal{L}_A(P)$ whenever P can do them, or may not, but will never happen so fast as to exclude the rest of P's interface and cause divergence. The abstracted user may even behave like $STOP$ and never do anything. Lazy abstraction has proved a powerful tool for formulating specifications such as security and fault tolerance. It is used in the CSP models of embedded systems created by Verum (see Section 6). Hiding corresponds to *eager* abstraction, as abstracted actions are always on offer to the process.

Some of Verum's modelling needed an abstraction that was somewhere between the two. They needed a model where the owner of A could still delay the process P, and still did not prevent other users getting to use P, but which would not refuse events in A for ever. So we might characterise these agents as *non-urgent* (as in lazy abstraction) but *ultimately compliant*. One class of actions that fits well into this model are ones (like the *tock* action described in [12, 13]) that represent the regular passage of time.

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 326–345, 2013.

We characterise *slow* abstraction $\mathcal{S}_A(P)$ by looking at limiting behaviour along infinite execution paths of P. While this abstraction proves impossible to express directly using CSP operators, we discover a method using priority for deciding whether $\mathcal{S}_A(P)$ refines a chosen specification. Beginning with a background section on CSP and its models, the rest of this paper develops the above ideas and ends with a case study showing how our methods have been used in Verum's ASD:Suite, a tool for creating correct-by-design embedded software.

We make two simplifying assumptions. Firstly we do not consider abstractions of processes able to terminate ($\checkmark$), avoiding some special cases. Secondly we only consider the case where the alphabet Σ is finite, though we will feel free to extend it, for modelling and analytic purposes, to a larger finite set.

2 Background

2.1 CSP and Its Semantics

CSP is based on instantaneous actions handshaken between a process and its environment, whether that environment consists of processes it is interacting with or some notional external observer. It enables the modelling and analysis of patterns of interaction. The books [7, 12, 13, 15] all provide thorough introductions to CSP. The main constructs that we will be using in this paper are set out below.

- The processes $STOP$, $SKIP$ and **div** respectively do nothing, terminate immediately with the signal $\checkmark$ and diverge by repeating the internal action τ. Run_A and $Chaos_A$ can each perform any sequence of events from A, but while Run_A always offers the environment every member of A, $Chaos_A$ can nondeterministically choose to offer just those members of A it selects, including none at all.
- $a \rightarrow P$ *prefixes* P with the single communication a which belongs to the set Σ of normal visible communications. Similarly $?x : A \rightarrow P(x)$ offers the choice A and then behaves accordingly.
- CSP has several *choice* operators. $P \,\Box\, Q$ and $P \sqcap Q$ respectively offer the environment the first visible events of P and Q, and make an internal decision via τ actions whether to behave like P or Q.
  The asymmetric choice operator $P \rhd Q$ offers the initial visible choices of P until it performs a τ action and opts to behave like Q. In the cases of $P \,\Box\, Q$ and $P \rhd Q$, the subsequent behaviour depends on what initial action occurs.
- $P \setminus X$ (hiding) behaves like P except that actions in X become τs.
- $P[\![R]\!]$ (renaming) behaves like P except that when P performs an action a, the new process performs some b that is related to a under the relation R.
- $P \underset{A}{\parallel} Q$ is a *parallel* operator under which P and Q act independently except that they have to agree (i.e. synchronise or handshake) on all communications in A. A number of other parallel operators can be defined in terms of this, including $P \,|||\, Q = P \underset{\emptyset}{\parallel} Q$ in which no synchronisation happens at all.

Other CSP operators such as $P;\ Q$ (sequential composition), $P \triangle Q$ (interrupt) and $P \,\Theta_a\, Q$ (throw an exception) do not play a direct role in this paper.

We understand a CSP process in terms of its pattern of externally visible communications. CSP has several styles of semantics that can be shown to be appropriately consistent with one another [12, 13]. The two styles that will concern us are *operational* semantics, in which rules are given that interpret any closed process term as a labelled transition system (LTS), and *behavioural* models, in which processes are identified with sets of observations that might be made from the outside.

An LTS models a process as a set of states that it moves between via actions in Σ^τ, where τ cannot be seen or controlled by the environment. There may be many actions with the same label from a single state, in which case the environment has no control over which is followed. The best known behavioural models of CSP are based on the following. *Traces* are sequences of visible communications a process can perform. *Failures* are combinations (s, X) of a finite trace s and a set of actions that the process can refuse in a *stable* state reachable on s. A state is stable if it cannot perform τ. *Divergences* are traces after which the process can perform an infinite uninterrupted sequence of τ actions, in other words diverge. The models are then

- $\mathcal{T}$ in which a process is identified with its set of finite traces;
- $\mathcal{F}$ in which it is modelled by its (stable) failures and finite traces;
- $\mathcal{N}$ in which it is modelled by its sets of failures and divergences, both extended by all extensions of divergences: it is *divergence strict*.

Traces, failures and divergences are all observations that can be made of a process in *linear time*. As described in [13], there is a range of other models based on richer forms of linear observation. An example is *refusal testing*, in which we record not just one stable refusal at the end of a trace, but have the option to record one before each event of the trace as well as at the end. Refusal testing models have long (see [10]) been recognised as being relevant to priority. However, we show in this paper that (unexpectedly) refusal-testing models are not always sufficient to encapsulate priority, and that sometimes one needs to look at the yet more refined models in which the refusal information during and at the end of traces is replaced by *acceptance* or *ready* sets: the actual sets of events made available from stable states. The latter are sometimes called *acceptance traces* models.

2.2 Lazy Abstraction

Lazy abstraction $\mathcal{L}_A(P)$ captures what a process looks like to an observer unable to see the events A, assuming that there is another user who can, and can accept or refuse them. See [12] for a full discussion.

The traces of $\mathcal{L}_A(P)$, like any way of abstracting A, are those of $P \setminus A$. As the abstracted user can at any time refuse or accept events in A, its failures are those of

$$(P \underset{A}{\parallel} \mathit{Chaos}_A) \setminus A$$

whose traces are again correct. We assume that the abstracted user cannot consume so much of P's resources as to make it diverge, so we assert that the abstraction (unlike the above CSP expression) never diverges.

2.3 FDR

FDR[11–13] is a refinement checker between finite-state processes defined in CSP. First created in the early 1990's it has been regularly updated since, and indeed a completely new version FDR3 will be released late in 2013.[1]

It uses CSP_M, namely CSP extended by Haskell-like functional programming. Thus one can define complex networks and data operations succinctly, and create functions that, given abstract representations of structures or systems, can automatically generate CSP networks to implement and check them. The FDR is at the heart of the verification functionality of ASD:Suite [3, 4]: the tool captures state machine models of proposed embedded systems, and then builds CSP models of how it will implement these so that they can be checked for correctness properties. This is only one of several major uses of FDR in government and industry, almost all of which start by translating some other notation to CSP.

FDR checks refinements of the form $Spec \sqsubseteq_X Impl$, where *Spec* is a process representing a specification in one of the standard CSP models X, usually traces, stable failures or failures-divergences. *Impl* is a CSP representation of the system being checked. Typically this sort of check scales better in *Impl* than *Spec*, the latter of which has to be normalised as part of the decision process. FDR supports a number of techniques for attacking the state explosion problem, including hierarchical compression. The algorithms underpinning FDR are set out in [12–14]. A number of recent additions to FDR including priority were summarised in [1].

3 A Priority Operator

There have been a number of proposals, for example [5, 8, 9], for the introduction of priority into CSP. These usually proposed ways in which a process could prefer one action to another, even though both remained available. An approach like that would automatically invalidate not only CSP's existing semantic models, which would have to be redeveloped to accommodate these preferences, but also the use of FDR in anything close to its usual form, since FDR supports transition systems without preferences. However, [13] introduced a priority that does make sense over ordinary labelled transition systems. The one we discuss here is a slightly more expressive version of that.

[1] The initial release of FDR3 will, functionally, be similar to FDR2.94 except that it will support multi-core execution of some functions, will have a new GUI, and will have an integrated type-checker for CSP_M. Further functionality is planned for later versions.

Our operator $\mathbf{Pri}_{\leq}(P)$ is parameterised by a partial order $\leq$ on Σ^{τ}, the set of action labels. τ is constrained to be maximal in $\leq$, but not necessarily maximum. (So there may be visible actions less than τ and ones incomparable to τ, but not ones greater than τ.) Further, we do not permit non-maximal elements of Σ to be incomparable to τ. These conditions are both required to preserve the property that CSP treats the processes P and $\tau.P$ (one that can perform a τ before acting like P) as equivalent.

The operational semantics of $\mathbf{Pri}_{\leq}(\cdot)$ are easier to understand than its abstract behavioural semantics. They do not, however, fit into the framework described as "CSP-like" in [13], because they require negative premises: an action cannot occur unless all actions of higher priority are impossible. The only operational rule needed for the new operator is

$$\frac{P \xrightarrow{x} P' \wedge \forall\, y \neq x . x \leq y . P \stackrel{y}{\not\longrightarrow} \cdots}{\mathbf{Pri}_{\leq}(P) \xrightarrow{x} \mathbf{Pri}_{\leq}(P')}$$

The fact that τ is maximal means it is never blocked by this rule.

Since the operational semantics for $\mathbf{Pri}_{\leq}(P)$ fall outside the "CSP-like" class that guarantees a semantics in every CSP model, it is not a surprise that not all such models are congruences for it. We cannot expect it to respect a model that does not tell us which events a process performs happen from stable states, and whether all Σ-events less than a given event are then refused. The traces model certainly does not tell us this because its observations are completely independent of whether the process is stable or not. While failures-based models would seem to satisfy this requirement – as failures occur in stable states and tell us what these states refuse – they do not give enough information. Consider the pair of processes $(a \rightarrow b \rightarrow STOP) \rhd (a \rightarrow STOP)$ and $(a \rightarrow STOP) \rhd (a \rightarrow b \rightarrow STOP)$. These divergence-free processes have identical failures, but imagine applying a priority operator to them where $a < b$. In each case, the $a \rightarrow \cdot$ that appears to the left of $\rhd$ is prevented because τ (necessarily, given our assumptions, of higher priority than a) is an alternative. So only the other a is allowed, meaning that the results of the prioritisation are different: one can perform b and one cannot. We conclude that it is not enough to know information about stable states only at the ends of traces; we also need to know about stability and the refusal of high-priority events earlier in traces.

The refusal-testing models do distinguish these two processes, because they have different behaviours after the refusal of all events other than a, followed by the action a have both been observed (which is written $\langle \Sigma \setminus \{a\}, a \rangle$ in the notation below). Several variations on the refusal-testing model, and a richer one in which exact *ready* or *acceptance* sets are recorded on the stable states in a trace, are detailed in Chapters 11 and 12 of [13]. In the simplest of these, the *stable refusal-testing model* $\mathcal{RT}$, the behaviours recorded of a process are all of the form

– $\langle X_0, a_1, X_1, \ldots, X_{n-1}, a_n, X_n \rangle$

where $n \geq 0$ and each X_i is either a refusal set (subset of Σ) or $\bullet$ (indicating that no refusal was observed).

The refusal-testing value of a process P can tell us what traces are possible for $\mathbf{Pri}_{\leq}(P)$: P can only perform an action a that is not maximal in $\leq$ when all greater actions (including τ) are impossible. In other words the trace $\langle a_1, \ldots, a_n \rangle$ is possible for $\mathbf{Pri}_{\leq}(P)$ if and only if

$$\langle X_0, a_1, X_1, \ldots, X_{n-1}, a_n, \bullet \rangle$$

is a refusal-testing behaviour, where X_i is $\bullet$ if a_{i-1} is maximal, and $\{a \in \Sigma \mid a > a_{i-1}\}$ if not (even if that set is empty so a_{n-1} is less than only τ).

It came as a surprise to us, however (particularly given what one of us wrote in [13]), to discover that there are cases where the refusal components of refusal-testing behaviours of $\mathbf{Pri}_{\leq}(P)$ can not be computed accurately from the corresponding behaviour of P. This is because $\mathbf{Pri}_{\leq}(P)$ can refuse larger sets than P: notice that if P offers *all* visible events, then the prioritised process refuses all that are not maximal in $\leq$. Consider the processes

$$DF1(X) = \bigsqcap \{a \to DF1(X) \mid a \in X\}$$

$$DF2(X) = \bigsqcap \{?x : A \to DF2(X) \mid A \subseteq X, A \neq \emptyset\}$$

These are equivalent in the refusal-testing models: each has all possible behaviours with traces in Σ^* that never refuse the whole alphabet Σ.

Now consider $Q1 = DF1(\{a, b\}) \,|||\, CS$ and $Q2 = DF2(\{a, b\}) \,|||\, CS$ where $CS = c \to CS$. Clearly $Q1$ and $Q2$ are refusal-testing equivalent. Let $\leq$ be the order in which $b > c$ and a is incomparable to each of b and c. We ask the question: is $\langle \{c\}, a, \bullet \rangle$ a refusal-testing behaviour of $\mathbf{Pri}_{\leq}(Qi)$?

When $i = 1$ the answer is "no", since whenever $Q1$ performs the event a the set of events it offers is precisely $\{a, c\}$. It can also offer $\{b, c\}$, but in neither case can it perform a after the refusal of $\{c\}$. However, $Q2$ can choose to offer $\{a, b, c\}$: in this state the priority operator prevents c from being offered to the outside, meaning that $\mathbf{Pri}_{\leq}(P2)$ can be in a stable state where a is possible but c is not: so in this case the answer is "yes". Thus we need more information than refusal testing of Qi to calculate the refusal-testing behaviours of $\mathbf{Pri}_{\leq}(Qi)$.

This example tells us that $\mathbf{Pri}_{\leq}(\cdot)$ can only be compositional for refusal testing when the structure of $\leq$ is such that whenever a and b are incomparable events in Σ and $c < b$ then also $c < a$. This is because we could reproduce an isomorphic example in any such order. It is, however, possible to give a compositional semantics for refusal testing when there is no such triple. This means that the order has to take one of two forms:

- A list of sets of equally prioritised events, the first of which contains τ.
- A list of sets of equally prioritised events, the first of which is exactly $\{\tau\}$, together with a further set of events that are incomparable to the members of the first two sets in the list and greater than those in the rest.

The priority order used in enforcing maximal progress in timed models does satisfy the above, but the $\leq$s we will use in analysing slow abstraction below do not satisfy it.

These issues disappear for the acceptance traces model $\mathcal{FL}$ and its variants, which are therefore the *only* CSP models with respect to which our priority operator can be defined in general. With respect to $\mathcal{FL}$, the semantics of $\mathbf{Pri}_{\leq}(P)$ are the behaviours (the A_i being stable acceptances or $\bullet$):

$$\{\langle A_0, a_1, A_1, \ldots, A_{n-1}, a_n, A_n\rangle \mid \langle Z_0, a_1, Z_1, \ldots, Z_{n-1}, a_n, Z_n\rangle \in P\}$$

where for each i one of the following holds:

- a_i is maximal under $\leq$ and $A_i = \bullet$ (so there is no condition on Z_i except that it exists).
- a_i is not maximal under $\leq$ and $A_i = \bullet$ and Z_i is not $\bullet$ and neither does Z_i contain any $b > a_i$.
- Neither A_i nor Z_i is $\bullet$, and $A_i = \{a \in Z_i \mid \neg\, \exists\, b \in Z_i . b > a\}$,
- and in each case where $A_{i-1} \neq \bullet$, $a_i \in A_{i-1}$.

3.1 FDR Implementation of Priority

The nature of the operational semantics of $\mathbf{Pri}_{\leq}(\cdot)$, in particular its use of negative premises, means that this operator cannot be folded into the *supercombinator* structures (see [13]) that lie at the heart of FDR's state machine implementation. It has therefore been implemented as a stand-alone operator that both inputs and outputs an LTS.

We decided that the practical version would have an easier-to-use input format rather than making all users construct a representation of a partial order with the constraints stated earlier. The implemented version therefore restricts the orders to ones that can be represented as a list of sets of visible events, where the first (the events incomparable to τ) may be empty:

```
prioritise(P,As)
```

where `As = <A0,A1...,An>` is a list of subsets of Σ.

These sets of events have lower priority as the index increases, so `An` are the ones of lowest priority. Importantly, there is no need for the `Ai` to cover all the visible events of `P`: those not in one of the `Ai` are incomparable to all other events including τ and neither exclude nor are excluded by any other.

3.2 Priority and Compression

FDR implements a number of operators that take an LTS and attempt to construct a smaller LTS or *Generalised* LTS (GLTS) with the same semantic value. A GLTS is like an LTS except that information such as divergence, refusals and acceptances may be included as explicit annotations to nodes rather than being deduced only from transitions.

With the exception of strong bisimulation, none of the compressions described in [14] is guaranteed to preserve the refusal-testing and acceptance traces models of CSP. In consequence, they cannot be reliably used inside a `prioritise` operator.

In part as a remedy for this problem, we have recently implemented the compression *divergence-respecting weak bisimulation* as defined in [13]. (This factors an LTS by the maximum weak bisimulation relation that does not identify any pair of states, one of which is immediately divergent and the other one not). This respects all CSP models and has the added advantage that, unlike some other compressions, it turns an LTS into another LTS rather than a GLTS. We will report separately on this implementation and weak bisimulation's place in the family of CSP compression functions.

4 Slow Abstraction

In the introduction we set out, informally, the problem of formulating the correct abstraction of a process P relative to an unseen user who is assumed to be lazy but eventually compliant with requests from the process, and who controls some subset A of P's events. We do not want these events to be visible to the external observer, as represented by our specification, which is expressed in the failures model $\mathcal{F}$ – a choice we will justify shortly. We will assume that P itself is free from divergence.

The abstracted events do not happen quickly enough for them to exclude offers of other events being accepted by the process indefinitely. Our model, therefore, is that the abstracted process has the *stable* failure (s, X) if and only if $(s, X \cup A)$ is a failure of P for some s' with $s' \setminus A = s$. We say it has the *unstable* failure (s, X) if and only if P has an infinite behaviour

$$P = P_0 \xrightarrow{x_0} P_1 \xrightarrow{x_1} P_2 \ldots$$

where, if u is the necessarily infinite trace consisting of all the non-τ x_i, $u \setminus A = s$ and there is some k such that (i) $x_i \in A \cup \{\tau\}$ for $i > k$, (ii) sufficient of the P_i are stable, an issue we will discuss below, and (iii) all but finitely many of these stable P_i refuse X. In other words it ultimately performs an infinite number of A events from stable states, all of which refuse X. We characterise the second sort as unstable failures because the abstraction is turning A actions into a sort of slow internal action, meaning that the refusals are occurring over a series of states linked by these actions.

The stable failure case, of course, corresponds to the regular definition of $P \setminus A$ over $\mathcal{F}$. The unstable case comes when the trace s in $P \setminus A$ is followed by an infinite sequence of events in $\{\tau\} \cup A$. Infinitely many of these must be in A because P is divergence-free.

Our idea that the abstracted user is lazy has to be made a little more specific here. It should not be too hard to see that this is closely related to the question of how many of the A actions in the trace above happen from stable states. Since P is divergence-free, it will always reach a stable state if left alone and so it is reasonable (though perhaps debatable if long finite sequences of τs can occur) to describe an A action that occurs from an unstable state as eager: our imaginary user has performed it before P's available τs had completed.

If all of these stable states, beyond a certain point, refuse X then we want (s, X) to be a failure of our abstraction.

- If all but finitely many of the As were eager, then it would neither make sense to describe our user as lazy, nor to detect any infinite pattern of refusals from the sequence of states: refusals will only happen from stable P_i. We assume that our user is too lazy for this to happen.
- If we insisted that all As happened in stable states, then this is quite a strong assumption about the user. States of P reachable only using eager occurrences of A would not be reached at all.
- The remaining two possibilities are that we insist (i) that only finitely many As are eager or (ii) less restrictively, that an infinite number of them are not eager. Each of these is a reasonable view, but we adopt (i) in part because we have a solution to the problem of automating it in FDR, but not (ii). So what we are saying is that the abstracted A-user is eventually sufficiently lazy not to take an event from an unstable state. The difference between (i) and (ii) shows up in the process

  $$Q = (a \to a \to Q) \rhd ((a \to a \to Q) \Box b \to STOP)$$

  If we abstract $\{a\}$ under assumption (i) then the result cannot refuse $\{b\}$ on $\langle\rangle$. However, under (ii) it can, because it would consider the behaviour in which every odd-numbered a occurs from Q's unstable initial state.

If an infinite, non-divergent behaviour of P has only finitely many visible actions outside A, and satisfies (i) we will call it $*A$-stable. We will spend much of this paper analysing such behaviours. For the process Q defined above, only finitely many as can occur from the initial state in a $*A$-stable behaviour.

An unstable failure only manifests itself over an infinite behaviour of P, which means that it would not make sense to ask what the process did *after* it. It would not, therefore, make sense to try to work out how our abstraction looks in a refusal-testing model. That is why in considering this type of abstraction we consider only failures specifications, given that for traces specifications we can just use $Spec \sqsubseteq_T P \setminus A$.

The value of $\mathcal{S}_A(P)$ is then the union of both its stable and unstable failures, paired with the traces of $P \setminus A$. This a member of $\mathcal{F}$, the "stable" failures model.

We have thus characterised what the behaviours of our new abstraction are, and so know what it means for it to refine some failures specification. This, however, gives us no clue about how to check properties of them in FDR.

It is interesting to compare $\mathcal{L}_A(P)$ and $\mathcal{S}_A(P)$. There is one direct relation that holds.

Lemma 1. *If P is a divergence-free process then $\mathcal{L}_A(P) \sqsubseteq_F \mathcal{S}_A(P)$.*

Proof. These two processes have the same set of traces by definition, so all we must do is show that every failure (stable or unstable) of $\mathcal{S}_A(P)$ belongs to $\mathcal{L}_A(P)$. This is trivially true for the stable ones, which are just those of $P \setminus A = (P \underset{A}{\parallel} Run_A) \setminus A$ and $Run_A \sqsupseteq Chaos_A$.

If (s, X) is an unstable failure of $\mathcal{S}_A(P)$ then, from definitions above it follows that there is an infinite series of traces $t_0 < t_1 < t_2 < \ldots$ where P has the failure (t_i, X) for each of them, and $t_i \setminus A = s$ for all i. Any one of them, combined with the failure $(t_i \upharpoonright A, A)$ of $Chaos_A$, yields the (stable) failure (s, X) in $(P \underset{A}{\parallel} Chaos_A) \setminus A$. ∎

Consider the process $P = a \rightarrow P \,\Box\, b \rightarrow a \rightarrow P$. Because $P \setminus \{a\}$ can diverge, eager abstraction $P \setminus \{a\}$ does not make sense. $\mathcal{L}_{\{a\}}(P) = b \rightarrow Chaos_{\{b\}}$ since if the abstracted user stops performing as at any stage then bs are forced to stop also. $\mathcal{S}_{\{a\}}(P)$, on the other hand, is just $Run_{\{b\}}$ as the abstracted user will always eventually perform a, enabling another b if that is necessary.

5 Unstable Failures Checking via Priority

We want to find a way of checking whether $Spec \sqsubseteq_F \mathcal{S}_A(P)$. That this holds for the traces and stable failures of the right-hand side can be established by checking $Spec \sqsubseteq_F P \setminus A$. We will assume this has been done, meaning that we want to check that the unstable failures of $\mathcal{S}_A(P)$ also satisfy $Spec$. Doing this on FDR will mean that any counterexample will manifest itself as a divergence, since this is the only sort of infinite counterexample that FDR can produce.

This tells us immediately that this type of behaviour cannot be checked in the usual CSP language without priority, since in that language the divergences of any context $F(P)$ depend only on the traces and divergences of P, not on its failures.

It also tells us we cannot find a context $G_A[\cdot]$ such that $G_A[P]$ has the same failures in $\mathcal{F}$ (the stable failures model) as $\mathcal{S}_A(P)$. It will therefore not be possible to test that this abstraction refines a failures specification $Spec$ by checking a refinement with $Spec$ itself on the left-hand side.

We can conclude that checking the unstable failures aspect of $Spec \sqsubseteq_F \mathcal{S}_A(P)$, at least without extending the functionality of FDR, must take the form

$$LHS(Spec, P, A) \sqsubseteq_M RHS(Spec, P, A)$$

in which M represents a model sensitive to divergence, and where an operator such as $\mathbf{Pri}_{\leq}(\cdot)$ that falls outside traditional CSP is used. We will see later that we can take LHS (independent of $Spec$, P and A), to be $Chaos_\Sigma$, the most nondeterministic divergence-free process, and M to be failures-divergences.

Before handling general $Spec$ we will first show how to deal with the case that $Spec$ is the failures specification of deadlock freedom:

$$DF = \bigsqcap \{a \rightarrow DF \mid a \in \Sigma\}$$

$\mathcal{S}_A(P)$ meets this specification provided both the following hold:

- P is deadlock free.
- There is no $*A$-stable behaviour of P such that eventually no action outside A is ever offered from a stable state.

$\mathcal{S}_A(P)$ will satisfy this provided (i) P is deadlock free in the usual sense and (ii) P has no state from which there is an infinite sequence of events in $A \cup \{\tau\}$ where all the A's are from states offering only subsets of A.

Priority can tell us if there is such a sequence starting from P's initial state. Prioritise all events outside A over those in A. Then $\mathbf{Pri}_{\leq}(Q)$ can perform an infinite sequence of A events if and only if none of them is offered from the same state as a higher priority, non-A event in $\Sigma \cup \{\tau\}$. Thus $\mathbf{Pri}_{\leq}(P) \setminus A$ has divergence $\langle\rangle$ if and only if P can perform an infinite trace of A events where none of them is from a state where a non-A event is offered.

Proving divergence freedom of this process does not, however, prove that $\mathcal{S}_A(P)$ can never unstably refuse the whole alphabet. If $a \in A$ and $b \notin A$ then, for any n, the process $\mathcal{S}_A(NB(n))$ can unstably refuse Σ, where

$$NB(0) = a \rightarrow NB(0)$$
$$NB(n) = (b \rightarrow STOP) \,\Box\, (a \rightarrow NB(n-1)) \quad (n > 0)$$

is the process that performs an infinite sequence of as with b offered as an alternative to the first n. Clearly, for $n > 0$, $\mathbf{Pri}_{\leq}(NB(n))$ is equivalent to $b \rightarrow STOP$, so hiding a will leave it divergence free.

We can solve this problem and find the unstable refusal in $\mathcal{S}_A(NB(n))$ if we introduce a second copy of a by renaming it, say to a', and make it incomparable with both a and b in the priority order. So in particular it can happen even when a is prevented by priority.

$\mathbf{Pri}_{\leq}(NB(n)[\![a, a' / a, a]\!])$ can now perform any number of a' events whatever the value of n After a trace of n or more a' events, this prioritised process will also be able to perform a, which is excluded in the initial states. Therefore $\mathbf{Pri}_{\leq}(NB(n)[\![a, a' / a, a]\!]) \setminus \{a\}$ can diverge after sufficiently long traces of a's.

These divergences simply reflect $NB(n)$'s ability to perform an infinite trace of a's with only finitely many offers of b along the way: by the time a particular divergence appears there are *no* further offers available.

The construction $\mathbf{Pri}_{\leq}(NB(n)[\![a, a' / a, a]\!]) \setminus \{a\}$ does not in itself represent the abstraction $\mathcal{S}_A(NB(n))$, both because it has the ability to perform a' actions, and because deadlocks in the abstraction have become divergences. It does, however, show us how to use FDR to check for the abstraction being deadlock free. This method generalises as follows.

Theorem 1. *For the divergence-free process P, the abstraction $\mathcal{S}_A(P)$ contains no unstable failure of the form (s, Σ) if and only if $((P[\![a, a' / a, a \mid a \in A]\!]) \setminus A$ is divergence free. Here it is immaterial whether a' is a single event disjoint from*[2] *$\alpha P \cup A$ or a separate such event for every member a of A.*

Note that the process checked here has the same number of states as P: every state of P is reachable because of the role of a', but there is only one state of this construct for each of P.

We now seek to generalise the above to a method for deciding whether *Spec* $\sqsubseteq_F$ $\mathcal{S}_A(P)$ for arbitrary divergence-free P and failures specification *Spec*.

[2] αP is the set of events used by P.

Suppose *Spec* is a specification process that $P \setminus A$ trace refines. We are trying to decide if $S \sqsubseteq_F \mathcal{S}_A(P)$. We can assume that no event of $\alpha Spec$ belongs to A, because if there were such events we could rename A to achieve this.

If S is any process such that $\alpha S \subseteq \alpha Spec$ and $(\langle\rangle, \Sigma) \notin failures(S)$, we can define $NR(S)$ to be the set of those X that are subset minimal with respect to $(\langle\rangle, X) \notin failures(S)$. $NR(S)$ is nonempty because Σ is finite and $(\langle\rangle, \Sigma) \notin S$.

If S is a process that can deadlock immediately, let $NR(S) = \emptyset$.

Choose a new d that is outside $\alpha Spec \cup A$. (Note that $\alpha P \subseteq \alpha Spec \cup A$ because we are assuming that $Spec \sqsubseteq_T P \setminus A$.) For a set of refusals $R \neq \emptyset$, let

$$T(R) = \Box_{X \in R}\, d \rightarrow (?x : X \rightarrow DS)$$

and $T(\emptyset) = DS$, where $DS = d \rightarrow DS$. Note that $T(R) \underset{\alpha Spec}{\parallel} Q$, for Q a process such that $S \sqsubseteq_T Q$, can deadlock if and only if, when one of the sets $X \in R$ is offered to P when it has performed $\langle\rangle$, P refuses it. This parallel composition is therefore deadlock free if no member of R is an initial (stable) refusal of P. Now let

$$Test(S) = (?x : S^0 \rightarrow Test(S/\langle x \rangle)) \ \Box\ T(NR(S))$$

For Q such that $Spec \sqsubseteq_T Q$, the parallel composition $Test(Spec) \underset{\alpha Spec}{\parallel} Q$ is then deadlock free if and only if $Spec \sqsubseteq_F Q$, given that we know that $S \sqsubseteq_T P$: the composition can deadlock if and only if, after one of its traces s, P can refuse a set that S does not permit. In understanding this it is crucial to note that the ds of $T(NR(S))$, including the one in the initial state of $Test(S)$, can occur unfettered in the parallel composition because they are not synchronised with Q. The first d that occurs fixes the present trace as the one after which $Test(Spec)$ checks to see that a disallowed refusal set (if any) does not appear in Q.

Our construction turns any case where Q fails *Spec* into a deadlock. It is very similar to the "watchdog" transformation for the usual failures model set out in [6]. The main difference is that ours is constructed with no τ actions: the visible action d replaces τ.

Consider the case where Q is replaced by P. This has the additional events A which are not synchronised with $Test(Spec)$, so the combination $Test(Spec) \underset{\alpha Spec}{\parallel} P$ can only deadlock in states where $P \setminus A$ has a stable failure illegal for *Spec*.

We would similarly like unstable failures of $\mathcal{S}_A(P)$ to turn into unstable "deadlocks", namely unstable refusals of Σ, in $\mathcal{S}_A(Test(Spec) \underset{\alpha Spec}{\parallel} P)$. This is confirmed by the following result.

Theorem 2. *Under the assumptions above, including the one that P is divergence-free and $Spec \sqsubseteq_F P \setminus A$, $\mathcal{S}_A(P)$ has an unstable failure that violates Spec if and only if $\mathcal{S}_A(Test(Spec) \underset{\alpha Spec}{\parallel} P)$ has an unstable failure of the form (s, Σ). Furthermore $\mathcal{S}_A(P) \sqsupseteq_F Spec$ if and only if*

$$(\mathbf{Pri}_{\leq}((Test(Spec) \underset{\alpha Spec}{\parallel} (P[\![a, a' / a, a \mid a \in A]\!])))) \setminus A$$

is divergence-free.

Proof. The second equivalence follows from what we know once we observe that, since *Test* never performs any member of A and $\alpha Spec \cap (A \cup \{a'\}) = \emptyset$,

$$Test(Spec) \underset{\alpha Spec}{\parallel} (P[\![a, a'/a, a \mid a \in A]\!]) = (Test(Spec) \underset{\alpha Spec}{\parallel} P)[\![a, a'/a, a \mid a \in A]\!]$$

So we need just to establish the first equivalence. Any unstable failure of the form (s, Σ) in $Test(Spec) \underset{\alpha Spec}{\parallel} P$ arises from a $*A$-stable behaviour of this combination such that eventually no action outside A is ever offered, and necessarily where eventually no event, other than members of A and τ, occurs. Since $Test(Spec)$ has no τ or A actions, there is a point in the infinite behaviour beyond which this process performs no action, so all the subsequent actions of the parallel composition are performed by P alone, with $Test(Spec)$ left in some "terminal" state. Since the resulting unstable refusal is Σ, this terminal state must be one refusing the unsynchronised d. Therefore the state is one offering some X such that $(s', X) \notin failures(Spec)$, where s is the trace up to any point in the infinite behaviour beyond the one at which $Test(Spec)$ first reaches its terminal state and $s' = s \setminus (A \cup \{d\})$. It should be clear that from this point extra events may add to s but will not change s'.

Since the infinite behaviour witnesses the unstable refusal of Σ in the parallel combination, we can assume that the infinite tail of states in which all the stable ones refuse $\Sigma \setminus A$ starts beyond the point where $Test(Spec)$ reaches its terminal state. The sequence of corresponding states in P must all refuse the X that this terminal state is offering. $P \setminus A$ has by that point performed s', recalling that $(s', X) \notin Spec$. P's behaviour thus witnesses the unstable failure $(s,' X)$, meaning that $\mathcal{S}_A(P)$ does not satisfy *Spec*.

It should not be difficult to see that the reverse also holds: if there is a $*A$-stable behaviour of P witnessing an unstable failure (s, X) of $\mathcal{S}_A(P)$ that violates *Spec*, we can assume that X is $\subseteq$-minimal with respect to this. The trace $s^\frown\langle d\rangle$ therefore leads $Test(Spec)$ to a state that offers exactly X. The combination $Test(Spec) \underset{\alpha Spec}{\parallel} P$ then has a $*A$-stable behaviour in which, after a finite trace s' such that $s' \setminus (A \cup \{d\}) = s$, $Test(Spec)$ permanently offers X and all stable states of P refuse it, linked only by τ and A actions. This behaviour clearly witnesses an unstable failure with refusal Σ. ∎

We therefore have a general technique for deciding whether, for divergence-free P, $\mathcal{S}_A(P)$ meets an arbitrary failures specification with respect to unstable failures.

6 Availability Checking in Verum's ASD:Suite

This example inspired the formulation of slow abstraction and the creation of the decision procedure in terms of priority.

6.1 Background on ASD:Suite

Analytical Software Design (ASD) [3] is a software design automation platform developed by Verum[3] that provides software developers with fully automated formal verification tools that can be applied to industrial scale designs without requiring specialised formal methods knowledge of the user. ASD was developed for industrial use and is being increasingly deployed by customers in a broad spectrum of domains, such as medical systems, electron microscopes, semi conductor equipment, telecoms and light bulbs. Industrial examples using ASD, such as the development of a digital pathology scanner, can be found in [4].

ASD is a component-based technology: systems contain both ASD components and foreign components. An ASD component is a software component specified, designed, verified and implemented using ASD and is specified by:

1) An ASD interface model specifying the externally visible behaviour of a component and
2) an ASD design model specifying its inner working and how it interacts with other components.

Corresponding CSP models are generated automatically from design and interface models, and the ASD component designs are formally verified using FDR, though the CSP is not visible to the end user. While design models are complete and deterministic, interface models are abstract and frequently nondeterministic.

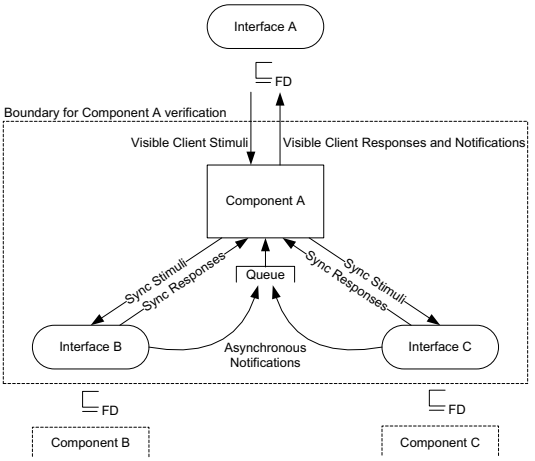


Fig. 1. ASD architecture

Figure 1 gives an overview of the standard ASD architecture which is based on the client-server model. Within an ASD model, system behaviour is specified in terms of stimuli and responses. A stimulus in Component *A* represents either

[3] `www.verum.com`

a synchronous procedure call initiated from a Client above or an asynchronous notification event received from its queue. A response in Component A will either be a response to its Client above or a synchronous procedure call downwards to Interfaces B or C.

The CSP model generated by ASD not only captures the behaviour in the models specified by the user, but also reflects the properties of the ASD runtime environment in which the generated code will be executed. This includes the externally visible behaviour of the foreign components and ASD components that form the environment in which the ASD design runs. Clients can initiate synchronous procedure calls to servers, with servers communicating in the other direction by return events and non-blocking queues.

The CSP models are verified for errors such as deadlocks, livelocks, interface non-compliance, illegal behaviour, illegal nondeterminism, data range violations, and refinement of the design and its interfaces with respect to a given specification. In Figure 1, the implemented interface is that Component A must satisfy is Interface A. As an example, a simplified version of the standard ASD timer interface specification is defined in Figure 2.

ITimer (ITimer.im)

ITimer | Application Interfaces | Notification Interfaces | Modelling Interfaces | Tags

SBS | States | State Variables

Timer already active - new timer can not be started

	Interface	Event	Guard	Actions	State Variable Updates	Target State	Comments	Tags
1	Inactive <>							
3	ITimer	CreateTimer(t)		ITimer.VoidReply		Active	ASD Timer started	
4	ITimer	CancelTimer		ITimer.VoidReply		Inactive		
6	Active <ITimer.CreateTimer(t)>							
8	ITimer	CreateTimer(t)		Illegal		-	Timer already active - new timer can not be started	
9	ITimer	CancelTimer		ITimer.VoidReply		Inactive		
10	IHwClock	Timeout		ITimerCB.Timeout		Inactive	Timer timed out	

Fig. 2. ASD interface model

There are 2 canonical states defined in this interface model, namely Inactive and Active. In the Inactive state, this interface offers 2 synchronous procedure calls to its client represented by the stimuli ITimer.CreateTimer and ITimer.CancelTimer. If its client calls ITimer.CreateTimer then the interface immediately returns with the synchronous return event ITimer.VoidReply, thereby passing the thread of control back to its client; the client is now free to carry on executing its own instructions and the interface is now in state Active. In state Active, there is a modelling event called IHwClock that represents the internal clock triggering an asynchronous notification event, ITimerCB.Timeout, to be put on its client's queue. This modelling event is hidden from its client reflecting the fact that the client cannot see the internal workings of the timer component and therefore doesn't know when it has occurred. Since the client's queue is non-blocking, from its client's point of view the interface might still be in Active or have moved to Inactive with a notification being placed on its queue. The modelling events can also be used to capture a nondeterministic choice over a range of response sequences that depend on internal behaviour abstracted from

the interface specification. Typically, a user will select whether modelling events are *eager*, namely that they will always occur if the system waits long enough for them, or *lazy* capturing the case where they nondeterministically might or might not occur. These correspond to the two main modes of abstraction for CSP described earlier, which play an important role in formulating ASD's CSP specifications.

A design model with its used interface models and appropriate plumbing, referred to as the *complete implementation*, is refined against its corresponding *implemented interface specification*, which specifies the design's expected visible behaviour by its client. In turn, this implemented interface becomes the used interface when designing and verifying the client component using it. In this refinement, the communication between the design model and its used interface models is hidden, since it is not visible to a client in this design. One of the properties that the complete implementation must satisfy is livelock freedom. For example, if a design can invoke an infinite cycle of communication with one or more of its used interfaces without any visible communication being offered to its client, we say the client is starved: this erroneous behaviour must be flagged and corrected. Within CSP such behaviour is captured as divergence.

6.2 Benign and Malign Divergence

There are divergences that arise during the verification of ASD models that are not regarded as erroneous behaviour in practice due to assumptions of fairness in the notion of 'time passing' at run-time. These are referred to as benign divergences.

An example of how a benign divergence arises in ASD is with the implementation of a timer driven polling loop as follows. An ASD component A is designed to monitor the state of some device by periodically polling it to request its status. In the event that the returned status is satisfactory, component A merely sets a timer, the expiry of which will cause the behaviour to be repeated. In the event that the returned status is not satisfactory, an asynchronous notification is sent to A's client and the polling loop terminates. Thus, A is not interested in normal results; it only communicates visibly to its client if the polled data is abnormal. Whenever component A is in a state in which it is waiting for the timeout event to occur, it is also willing to accept client API stimuli, one of which may be an instruction to stop the polling loop. The design of component A has at least 2 used interfaces, one of them being the Timer interface described above, and the other being the interface, PolledUC, for the used component whose status is being polled. This is summarised in Figure 3.

A subset of the behaviour of the design of component A relevant to this discussion can be summarised by the state transition diagram in Figure 4. The events prefixed with CLIENT represent the communication that is shared with the specification on the left-hand side of the refinement and therefore remains visible; all the other events become hidden. The labelled states represent the states of interest for the purposes of describing the divergence in question. All event labels are prefixed with the component name that shares the communication with the

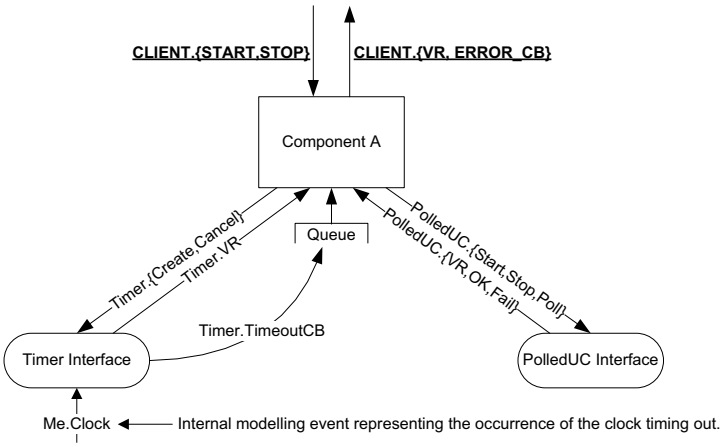


Fig. 3. Component A and its interfaces

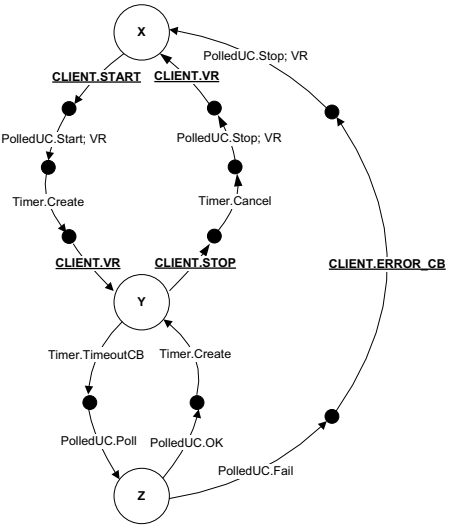


Fig. 4. Subset of component A's behaviour

design. Events with labels ending in CB are asynchronous notification events that are taken from the design's queue. The divergence occurs in state Y, where the system can perform an infinite cycle of hidden events via state Z, repeating the action of timing out, discovering that the polled component is fine and setting the timer again.

In the CSP model and at run-time, A could carry on polling device PolledUC indefinitely. However, at run-time a distinction is made between τ loops where a client API call is available as an alternative and τ loops that offer no alternative and therefore genuinely starve the client. In the former case, the design's client is able to intervene and perform a procedure call that breaks this loop. Provided such a client API stimulus is available, this divergence is not regarded as an error in the design; it will not diverge at run-time because in the real environment time passes between creating a new timer and the corresponding timeout notification event, during which the client is able to perform an API call. The design is correct under that assumption which can be safely made due to the implementation of the Timer component. In the example design in the diagram above, the visible event CLIENT.STOP is available in state Y as an option for breaking the diverging cycle of τ events. The assumption at run-time is that the internal clock does not timeout instantaneously, assuming that the create timer procedure call did not set the timer to 0. It is also assumed that it will eventually occur. Therefore a client using the timer process can rely on its occurrence as well as there being some time that passes within which the client may legitimately communicate with components above it in the stack (i.e. the client's client).

For this we use our new *slow* abstraction for the modelling event Me.Clock rather than eager or lazy, which would not be correct. Prior to the discover of this technique, the only option was to place artificial assumptions in for form of restraints upon the occurrence of such modelling events. This both increased the state space and carried the risk of missing erroneous behaviours during verification.

One can describe divergences composed of abstracted events, during which the slow abstraction makes an offer, as being *benign*, whereas ones where the offers end, or which are composed of other hidden events, as being *malign* and genuinely erroneous. Our new methods allow this distinction to be made.

The analysis in ASD uses the priority-based techniques described in Section 5. The set of modelling events M is partitioned into two sets. The first set M_{SE} comprises the slow eager modelling events that are controlled by the external used components and are assumed to occur eventually, but not so fast that their speed starves their client, for example ME.Clock in the timer polling loop example described above. The second set M_L comprises the modelling events that might or might not occur and are therefore accurately modelled by lazy abstraction.

If P is the system model with all these modelling events left visible, a divergence in $P \setminus M_{SE}$ can take three forms:

- The infinite sequence of τs may only contain finitely many hidden M_{SE} actions. This clearly represents a form of malign divergence.
- There might be infinitely many hidden M_{SE} actions, only finitely many of which have the alternative of a client API event. This is another form of malign divergence since there is the possibility of client starvation.
- Finally, infinitely many of the M_{SE} events might have a client API event as an alternative. As discussed above, this is a benign divergence.

You can think of there being a distinction between "slow τs" formed by hiding M_{SE} – these give the client time to force an API – and ordinary "fast τs", which do not. This is just the view formed by the slow abstraction of the events mapping to the slow τs after conventional hiding (eager abstraction) of the ones mapping to the fast ones.

Checking the divergence-freedom of

$$\mathbf{Pri}_{\leq}(P[\![m, m' / m, m \mid m \in M_{SE}]\!]) \setminus M_{SE}$$

gives precisely the check for malign divergence that we want: it does not find benign ones. If we needed to check more precisely what API offers were made along sequences of M_{SE} events, we could use the machinery of unstable failures checking discussed earlier in this paper.

After establishing that all divergences are benign, and if necessary make correct offers, the rest of the system properties can be checked in the stable failures model of CSP, as is conventional for checks involving lazy abstraction. The ASD use of slow abstraction described here corresponds exactly to the case of checking for deadlock freedom which was the core case earlier.

7 Conclusions

We have explained the CSP priority operator in terms of operational semantics and shown that, depending on the form of the partial order used, it requires either CSP's refusal-testing or acceptance traces model for compositionality.

We have also studied the problem of abstracting an interface that is neither eager nor is allowed to be completely idle, capturing refusal information from infinite traces. The reason for studying this alongside priority is that priority was the key to automating checks of the *slow* abstraction we developed against failures specifications.

Our industrial case study was satisfying because this was an example in which a practical problem inspired the creation of a piece of theory (i.e. slow abstraction and the priority technique for checking properties of it) that would not have been discovered without it. Beyond the scope of the present paper, we have had to bring further fairness considerations into our models to handle further nuances of the ASD models. That will be the subject of a future paper.

The techniques developed in this paper are applicable wherever one models a system which has events that, either because of their assumed internal control or the way they are assumed to be controlled by an unseen external agent, progress

in a measured rather than eager manner. It would be interesting to investigate the relationship with Schneider's theory of *timewise refinement* [15], which shows how Timed CSP processes can be seen to satisfy untimed specifications: at least in discretely timed versions of CSP, this appears to be closely related to the slow abstraction of time. Clock and time signals, in general, appear to be excellent candidates for this form of abstraction.

References

1. Armstrong, P., Goldsmith, M., Lowe, G., Ouaknine, J., Palikareva, H., Roscoe, A.W., Worrell, J.: Recent developments in FDR. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 699–704. Springer, Heidelberg (2012)
2. Armstrong, P., Lowe, G., Ouaknine, J., Roscoe, A.W.: Model checking Timed CSP. To appear in Proceedings of HOWARD, Easychair
3. Hopcroft, P.J., Broadfoot, G.H.: Combining the box structure development method and CSP. Electr. Notes Theor. Comput. Sci. 128(6), 127–144 (2005)
4. Broadfoot, G.H., Hopcroft, P.J.: A paradigm shift in software development. In: Proceedings of Embedded World Conference 2012, Nuremberg, February 29 (2012)
5. Fidge, C.J.: A formal definition of priority in CSP. ACM Transactions on Programming Languages and Systems 15(4) (1993)
6. Goldsmith, M., Moffat, N., Roscoe, A., Whitworth, T., Zakiuddin, I.: Watchdog transformations for property-oriented model-checking. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 600–616. Springer, Heidelberg (2003)
7. Hoare, C.A.R.: Communicating sequential processes. Prentice Hall (1985)
8. Lawrence, A.E.: CSPP and event priority. Communicating Process Architectures 59 (2001)
9. Lowe, G.: Probabilistic and prioritised models of Timed CSP. Theoretical Computer Science 138(2) (1995)
10. Phillips, I.: Refusal testing. Theoretical Computer Science 50(3) (1987)
11. Roscoe, A.W.: Model checking CSP. In: A Classical Mind: Essays in Honour of C.A.R. Hoare. Prentice Hall (1994)
12. Roscoe, A.W.: The theory and practice of concurrency. Prentice Hall (1997)
13. Roscoe, A.W.: Understanding concurrent systems. Springer (2010)
14. Roscoe, A.W., Gardiner, P.H.B., Goldsmith, M.H., Hulance, J.R., Jackson, D.M., Scattergood, J.B.: Hierarchical compression for model-checking CSP *or* how to check 10^{20} dining philosophers for deadlock. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 133–152. Springer, Heidelberg (1995)
15. Schneider, S.A.: Concurrent and real-time systems: the CSP approach. Wiley (2000)

Performance Estimation Using Symbolic Data

Jian Zhang

State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing 100190, China
zj@ios.ac.cn, jian_zhang@acm.org

Abstract. Symbolic execution is a useful technique in formal verification and testing. In this paper, we propose to use it to estimate the performance of programs. We first extract a set of paths (either randomly or systematically) from the program, and then obtain a weighted average of the performance of the paths. The weight of a path is the number of input data that drive the program to execute along the path, or the size of the input space that corresponds to the path. As compared with traditional benchmarking, the proposed approach has the benefit that it uses more points in the input space. Thus it is more representative in some sense. We illustrate the new approach with a sorting algorithm and a selection algorithm.

1 Introduction

For an algorithm (or program), its efficiency is very important. This is often characterized by the worst-case or average-case complexity. But we may not know the exact complexity for every program we write. Sometimes the complexity of algorithms is difficult to analyze, even for experts. For example, the Boyer-Moore string matching algorithm was published in 1977 [1]. More than 10 years later, Richard Cole [2] showed that the algorithm performs roughly $3n$ comparisons.

Even when we know the complexity, the actual performance may be different. As a matter of fact, some polynomial-time algorithms do not have good performance in practice. For two algorithms, even though they both have polynomial-time complexity, their execution times may be different due to the difference in constant factors. Thus benchmarking or empirical evaluation is still necessary in most cases. We run the program for a certain number of times, usually with random input data; and analyze the results (e.g., execution times of the program).

In this paper, we propose a *symbolic benchmarking* approach, in which the program/algorithm is executed symbolically. A benefit of this approach is that one symbolic execution corresponds to many actual executions. The performance is measured in terms of execution times or some performance indicators (e.g., the number of comparisons of array elements).

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 346–353, 2013.

2 Basic Concepts and Related Works

A program (algorithm) can be represented by its flow graph, which is a directed graph. From the graph, one may generate many (or infinite number of) paths. Given a path, it is called *feasible* or *executable* if there is some input vector which makes the program execute along the path. Otherwise, it is called *infeasible* or *non-executable*. For a feasible path, there may be many different input data which make the program execute along the path. The input data can be characterized by a set of constraints, called the *path condition*. The path is feasible if and only if the path condition is satisfiable. Each solution of the path condition, when given as the initial input to the program, will make the program execute along the path.

Let us look at a simple example. The following C program computes the greatest common divisor (gcd) of two positive integers (`m` and `n`).

```
int GCD(int m, int n)
{
    x = m;              /* 1 */
    y = n;              /* 2 */
    while (x != y) {    /* 3 */
      if (x > y)        /* 4 */
        x = x-y;        /* 5 */
      else y = y-x;     /* 6 */
    }                   /* 7 */
    gcd = x;            /* 8 */
}
```

The following are three paths:

```
P1: 1-2-3-8
P2: 1-2-3-4-5-7-3-8
P3: 1-2-3-4-6-7-3-8
```

The first path can be represented as follows:

```
    x = m;
    y = n;
  @ !(x != y);
    gcd = x;
```

Here `@` denotes a condition (e.g., loop condition or assertion); and '`!`' means the negation of a condition.

The path conditions for the three paths are: (1) $m = n$; (2) $m = 2n$; (3) $n = 2m$, respectively. Obviously, given any pair of integers satisfying the equation $m = n$ as the initial input data, the program will run along path `P1`. And, if we provide any pair of positive integers m, n satisfying the second equation as the input to the program, the program will execute along path `P2`.

How can we obtain the path condition, given a path in the program? This can be done via *symbolic execution* [5]. In contrast to concrete execution, symbolic

execution takes symbolic values as input, e.g., $m = m_0$, $n = n_0$; and then simulates the behavior of the program. During the process, each variable can take an expression as its value, e.g., $x = m_0 + 2n_0$.

Combined with constraint solving, symbolic execution can be very helpful in static analysis (bug finding) and test case generation [10]. For such purposes, usually we need to find just one solution to each path condition. That solution may correspond to a test case, or the input vector which shows a bug in the program.

More generally, we may consider the number of solutions to a path condition. In [9], we proposed a measure $\delta(P)$ for a path P in a program. Informally, it denotes the number of input vectors (when the input domain is discrete) which drive the program to be executed along path P. In other words, it denotes the volume of the subspace in the input space of the program that corresponds to the execution of path P, or equivalently, the size of the solution space of the path condition.

For the GCD example, suppose that each of the input variables `m` and `n` can take values from the domain [1..100]. So there are 10,000 different input vectors (i.e., pairs $\langle m, n \rangle$). It is easy to see that

$$\delta(P1) = 100,\ \delta(P2) = 50,\ \delta(P3) = 50.$$

In [8], we studied how to compute the size of the solution space, given a set of constraints (where each constraint is formed from Boolean variables and linear inequalities using logical operators like negation, disjunction, conjunction). We developed a prototype tool for doing this [11]. For example, suppose the input formula is $(x \neq 0) \vee b$. Here x is an integer variable and b is a Boolean variable. If we use the command-line option `-v3` which specifies the word length of integer variables to be 3, the result is 15. The reason is that, when b is TRUE, any value of x can satisfy the formula; and when b is FALSE, any non-zero value of x satisfies the formula. So, there are $8 + 7 = 15$ solutions in total.

In [8], we also introduced a measurement called *execution probability*. Given a path P, its execution probability is $\delta(P)$ divided by the total volume of the involved data space. For path P1 in the GCD example, its execution probability is 0.01; For each of the other two paths, the execution probability is 0.005.

In [7], we proposed to move from qualitative analysis to quantitative analysis, which combines symbolic execution with volume computation to compute the exact execution frequency of program paths and branches. We argue that it is worthwhile to generate certain test cases which are related to the execution frequency, e.g., those covering cold paths.

Recently, Geldenhuys et al. [3] also discussed how the calculation of path probabilities may help program analysis (esp. bug finding). They implemented a prototype tool which can analyze Java programs.

The above works focus on finding bugs in programs, while the main purpose of the approach presented here is to estimate the performance of programs.

3 Performance Estimation

In practice, the performance of a program/algorithm is often measured by the running times on some machine, with respect to a set of benchmarks. This is useful. But the result often depends on the choice of machines and benchmarks.

Rather than using running times, we may also use some performance indicators or performance indices (PINDs in short) which can describe a program's performance on a family of machines. For instance, we may choose the number of memory accesses as a PIND, or the number of multiplication operations as a PIND. When studying sorting algorithms, we may choose the number of comparisons, or the number of swaps (i.e., element exchanges). Actually, Don Knuth [6] proposed to use the number of memory operations (*mems*) rather than execution times on some particular machine, when reporting the performance of some algorithm/program.

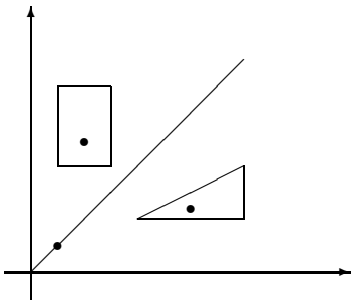
Given a program, we can estimate its performance by generating a set of program paths, denoted by $\{P_i\}$. For each path P_i, usually it is not difficult to determine the PIND value $pind_i$. The performance of the whole program can be computed by taking an average over the set of paths. The contribution of path P_i is weighted by its δ value δ_i, or its execution probability. More formally, the program's performance is estimated to be $\sum_i(\delta_i * pind_i)/\sum_i \delta_i$.

Of course, we can also extend the old-fashioned way. We may execute the program for a number of times (with different input data), remembering its actual performance (e.g., $time_1 = 5$ ms, $time_2 = 8$ ms), as with ordinary empirical evaluation. And then, for each concrete execution, we

- extract the path;
- get the path condition;
- compute the delta value of the path.

Finally, we get an estimation of the program's performance using the formula $\sum_i(\delta_i * time_i)/\sum_i \delta_i$.

Let us illustrate the approach/methodology using the following picture:



Here we assume that the input space is 2-dimensional (i.e., there are two input variables for the program, as in the case of GCD).

The above picture compares traditional benchmarking (examining the program's behavior at three points) with our approach (examining the program's

behavior with respect to three sets of input data: one line, one triangle, and one rectangle). If some set is too small (as compared with others), its contribution can almost be neglected.

4 Examples

4.1 Sorting

Sorting algorithms are very important in computer science, and they have been studied extensively. For many basic sorting algorithms, we can choose the number of comparisons (or the number of swaps) as a PIND.

Let us consider the following bubble sort algorithm.

```
for(i = 0; i < N-1; i++)
  for(j = N-1; j > i; j--) {
    if (a[j-1] > a[j])
      swap(a[j-1], a[j]);
  }
```

We assume that the input array `a` is of size 4. Then there are 24 *feasible* paths. For each path P_i, we can get the path condition and the value of $\delta(P_i)$.

Assume that the value of each array element is between 0 and 10 (i.e., $0 \leq a[i] < 10$). Using symbolic execution and our volume computation tool [8,11], we can find that the δ value of each path is almost the same (a little more than 416.66). In this sense, the example is a bit special.

For each path, the number of comparisons is the same, i.e., 6. However, the number of swaps is different, ranging from 0 to 6. The total number of swaps for all the 24 paths is 72. So, on average, the number of swaps performed by the bubble sort algorithm is 3 (when $N = 4$). In summary, when $N = 4$, the average performance of the above program is like the following: 3 swaps of array elements and 6 comparisons between array elements.

Note that the domain [0..10) is arbitrary. We can replace 10 by 100, for instance.

4.2 Selection

Now we consider the selection algorithm FIND [4]. It can be described in C-like language, as follows:

```
m = 0;
n = N-1;
while (m < n) {
  r = A[f];
  i = m; j = n;
  while (i <= j) {
    while (A[i] < r)
```

```
        i = i+1;
      while (r < A[j])
        j = j-1;
      if (i<=j) {
         w = A[i]; A[i] = A[j]; A[j] = w;
         i = i+1;
         j = j-1;
      }
    }
    if (f<=j) n = j;
    else if (i<=f) m = i;
    else {
      m = 1; n = 0;
    }
  }
```

Given an array of size N (denoted by `A[N]`) and a non-negative integer f, the algorithm tries to find the element whose value is the f'th in the array; and rearranges the array such that this element is placed in `A[`f`]`. At the end of the algorithm, the elements of the array should satisfy the following relationship:

$$A[0], A[1], \ldots, A[f-1] \leq A[f] \leq A[f+1], \ldots, A[N-1].$$

From the above source code, we can extract a number of paths from its flow graph. The following are several *short* paths:

Table 1. Paths in the FIND Program

pathID	nComp	nSwap	δ
w11	9	3	16303680
w16	9	2	12191040
w18	9	3	16303680
w20	9	2	12191040

Here `nComp` means the number of comparisons of array elements; and `nSwap` means the number of swaps of array elements. We assume that $f = 3$, and the size of the array is 8, i.e., $N = 8$. We also assume that each element of the array A is an integer, and it takes its value from a finite domain: $-8 \leq A[i] < 8$.

We note that the size of the whole input space is $16^8 = 2^{32}$. And the δ values in the above table are still very small, as compared with this size. But we believe that it is better to use these subspaces than a small set of single points (as with traditional empirical evaluation).

It is not true that all the paths are similar. For instance, we randomly generated the following two input vectors:

```
A[8] = { -2, 5, 6, 3, 1, 0, -7, 6 };
A[8] = { 2, 0, -2, -8, 4, -4, 5, 1 };
```

For the former, the execution path has 4 swap operations; but for the latter, the execution path has 6 swap operations.

We may execute the program for 2 times, with the above data as input. Then we track the execution traces and obtain two program paths. From each path, we can get the path condition using symbolic execution.

The path condition for the first case is the following:

```
 (A[0] < A[3]);  !(A[1] < A[3]);  (A[3] < A[7]);
!(A[3] < A[6]);  !(A[2] < A[3]); !(A[3] < A[5]);
!(A[3] < A[4]);   (A[0] < A[4]);  (A[6] < A[4]);
 (A[5] < A[4]).
```

And the δ value is 4075920. On the other hand, the δ value for the second case is just 87516. The corresponding path condition is the following set of constraints.

```
!(A[0] < A[3]);  (A[3] < A[7]);  (A[3] < A[6]);
 (A[3] < A[5]);  (A[3] < A[4]); !(A[1] < A[3]);
 (A[3] < A[2]);  (A[3] < A[1]);  (A[1] < A[0]);
 (A[2] < A[0]); !(A[0] < A[7]); !(A[4] < A[0]);
 (A[0] < A[6]); !(A[0] < A[5]);  (A[1] < A[7]);
 (A[2] < A[7]); !(A[7] < A[5]); !(A[1] < A[5]);
!(A[2] < A[5]);  (A[5] < A[2]);  (A[2] < A[1]);
```

If we take an average over the paths using the traditional method, the number of swaps will be $(4 + 6) = 5$. But if we take a weighted average, the number of swaps will be $(4075920 * 4 + 87516 * 6)/(4075920 + 87516)$, which is about 4.04.

5 Discussion on the Limitations

One obvious difficulty with the approach is the combinatorial explosion problem. In general, a program has many (or an infinite number of) execution paths. For most programs, it is impossible to examine all the paths. We can just randomly choose a small subset of them, or choose as many as we can. Thus the result is only an *estimation* of the overall performance of the program.

Currently symbolic execution and volume computation are still expensive. But we think that the approach is practical when applied to kernel algorithms (which are not too long in terms of lines of code) or models of software systems.

One assumption that we take is that all the points in the input data space are evenly distributed. This may not be the case in certain applications. In the future, we will consider other distributions.

Sorting and selection algorithms are special in that the major operations are comparison and object movements. For a sorting algorithm like bubble sort, its behavior can often be determined by considering all possible permutations of the input vector. When the size of the input array is a small fixed positive integer, we can exhaustively examine all the paths of the algorithm, and then obtain the accurate average PIND value. But in most cases, it is not easy (if not impossible) to do this.

6 Concluding Remarks

In this paper, we propose to calculate or estimate the average performance of algorithms (programs) automatically, by analyzing their feasible paths and using volume computation techniques. We report some experiments with several common algorithms.

It should be emphasized that the approach is general, in that it is not restricted to a particular algorithm (or some class of algorithms). The approach can be automated. In fact, there are already some automatic tools available. As compared with traditional benchmarking, the approach can be more complete (i.e., covering a larger part of the input space).

As mentioned in the previous section, there are still some challenges for the new approach. But we believe it offers a better way to estimate the performance of programs. In the future, we will tackle the challenges, so that the approach is more practical and applicable to many algorithms.

Acknowledgements. The author would like to thank Peng Zhang for his comments on an earlier version of this paper.

References

1. Boyer, R., Moore, S.: A fast string matching algorithm. Comm. ACM 20, 762–772 (1977)
2. Cole, R.: Tight bounds on the complexity of the Boyer-Moore string matching algorithm. In: Proc. of the 2nd Symp. on Discrete Algorithms (SODA 1991), pp. 224–233 (1991)
3. Geldenhuys, J., Dwyer, M.B., Visser, W.: Probabilistic symbolic execution. In: Proc. of the Int'l Symposium on Software Testing and Analysis (ISSTA 2012), pp. 166–176 (2012)
4. Hoare, C.A.R.: Proof of a program: FIND. Commun. ACM 14(1), 39–45 (1971)
5. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
6. Knuth, D.E.: The Stanford GraphBase: A Platform for Combinatorial Computing. ACM Press (1994), `http://www-cs-faculty.stanford.edu/~knuth/sgb.html`
7. Liu, S., Zhang, J.: Program analysis: from qualitative analysis to quantitative analysis. In: Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE 2011), pp. 956–959 (2011)
8. Ma, F., Liu, S., Zhang, J.: Volume computation for Boolean combination of linear arithmetic constraints. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 453–468. Springer, Heidelberg (2009)
9. Zhang, J.: Quantitative analysis of symbolic execution. Presented at the 28th Int'l Computer Software and Applications Conf. (COMPSAC 2004) (2004)
10. Zhang, J.: Constraint solving and symbolic execution. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 539–544. Springer, Heidelberg (2008)
11. Zhang, J., Liu, S., Ma, F.: A tool for computing the volume of the solution space of SMT(LAC) constraints, draft (January 2013)

Synthesizing Switching Controllers for Hybrid Systems by Generating Invariants*

Hengjun Zhao[1,2], Naijun Zhan[1], and Deepak Kapur[3]

[1] State Key Lab. of Comput. Sci., Institute of Software, CAS, Beijing, China
[2] University of Chinese Academy of Sciences, Beijing, China
[3] Dept. of Comput. Sci., University of New Mexico, Albuquerque, NM, USA
{zhaohj,znj}@ios.ac.cn, kapur@cs.unm.edu

Abstract. We extend a template-based approach for synthesizing switching controllers for semi-algebraic hybrid systems, in which all expressions are polynomials. This is achieved by combining a QE (quantifier elimination)-based method for generating invariants with a qualitative approach for predefining templates. Our synthesis method is relatively complete with regard to a given family of predefined templates. Using qualitative analysis, we discuss heuristics to reduce the numbers of parameters appearing in the templates. To avoid too much human interaction in choosing templates as well as the high computational complexity caused by QE, we further investigate applications of the SOS (sum-of-squares) relaxation approach and the template polyhedra approach in invariant generation, which are both supported by modern numerical solvers.

1 Introduction

Hybrid systems, in which computations proceed by continuous evolutions as well as discrete jumps simulating transitions from one mode to another mode, are often used to model devices controlled by computers in many application domains [1]. Combining ideas from state machines in computer science and control theory, formal analysis, verification and synthesis of hybrid systems have been an important area of active research. In verification problems, a given hybrid system is required to satisfy a desired safety property e.g. that the temperature of a nuclear reactor will never go beyond a maximum threshold, as it may cause serious economic, human and/or environmental damage, thus implying that the system will never enter any unsafe state. A synthesis problem is harder given that the focus is on designing a controller that ensures the given system will satisfy a safety requirement, reach a given set of states, or meet an optimality criterion, or a desired combination of these requirements.

Automata-theoretic and logical approaches have been primarily used for verification and synthesis of hybrid systems [2,4,45]. In [4,45], a general framework

* This work has been supported in part by the projects NSFC-91118007, National Science and Technology Major Project of China (Grant No. 2012ZX01039-004), NSF CCF 1248069 and NSF DMS 1217054.

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 354–373, 2013.

for controller synthesis based on hybrid automata was proposed. This approach relies on *backward reachable set* computation and *fixed point iteration*, and thus has two main restrictions: (i) the computation of backward reachable set is hard for most continuous dynamics, and (ii) termination of the fixed point iteration process cannot be guaranteed, even for those hybrid systems whose backward reachable sets are effectively computable. Therefore most of the research, e.g. [14], focuses on overcoming the above two restrictions.

Recently, a deductive approach for verification and synthesis based on constraint solving was proposed in [11,28,27,41,21,40]. The central idea is to reduce verification and synthesis problems of hybrid systems to invariant generation problems, much like verification of programs. As proposed in [16,15,34], if invariants are hypothesized to be of certain shapes, then corresponding templates with associated parameters can be used and the invariant generation problem can be reduced to constraint solving over parameters by quantifier elimination. This methodology is used in [41] for synthesizing switching controllers meeting safety requirements, while in [43], the approach is extended for satisfying both safety and reachability requirements. A common problem with template-based method is that it heavily relies on a user specifying the shape of invariants that are of interest, thus making it interactive and user driven, raising doubts about its scalability and automation. Besides, the inference rules for inductive invariants in [42,41,43] are sound and complete for several classes of invariants, e.g. smooth, quadratic and convex invariants, but are not complete for generic semi-algebraic sets[1].

Inspired by [17,4,41] and [22], we extend in this paper the template-based invariant generation approach for synthesizing switching controllers of hybrid systems to meet given safety requirements. The paper makes the following contributions:

- We propose a method for synthesizing switching controllers for hybrid systems using a family of invariants, which could be different for different modes of a hybrid system. We use templates for invariant generation based on quantifier-elimination (QE) techniques combined with numerical methods.
- In the QE-based synthesis approach, we adopt the invariant generation method proposed in [22], which is proved to be sound and relatively complete with respect to a given shape of semi-algebraic invariants (i.e. a given family of predefined semi-algebraic templates). The advantage is that, compared to the invariant generation methods used in [42,41,43], there is more possibility of discovering invariants of the given shape.
- Using the qualitative approach proposed in [17] for analyzing continuous evolution in each mode of a hybrid system, we can identify those continuous states at which even small perturbation would lead to continuous evolution violating the safety requirement. Such continuous states are called critical control points, using which we can determine a more precise shape of

[1] A subset $A \subseteq \mathbb{R}^n$ is called *semi-algebraic* if there is a quantifier-free polynomial formula φ s.t. $A = \{\mathbf{x} \in \mathbb{R}^n \mid \varphi(\mathbf{x}) \text{ is true}\}$.

templates to be used as invariants, thus reducing the number of parameters appearing in templates.
- Quantifier elimination techniques have high complexity especially for cases when templates have lots of parameters. Even though qualitative analysis is helpful in bringing down the number of parameters and thus the complexity of QE, the paper also explores two kinds of predefined templates where numerical techniques can be exploited to improve the degree of automation and scalability. In particular, (i) for polynomial templates, using *sum-of-squares* (SOS) relaxation, the constraint on parameters appearing in templates is transformed into a *semi-definite program*(SDP), which is convex and can be solved efficiently; (ii) for linear systems and a special type of templates—template polyhedra, the invariant generation problem can be reduced to a BMI (*bilinear matrix inequality*) feasibility problem, which is also easier to solve (numerically) than QE.

Paper Structure. The rest of this paper is structured as follows. In Section 2 we formally define the switching controller synthesis problem for safety of hybrid automata. In Section 3 we introduce the notion of invariant in the context of hybrid system, and formulate an abstract solution to the controller synthesis problem using invariants; then we extend a template-based method for invariant generation to solve the controller synthesis problem, by combining quantifier elimination (QE) techniques and qualitative analysis. In Section 4 we investigate the application of two numerical approaches in switching controller synthesis by generating invariants numerically. We finally conclude the paper with some discussions by Section 5.

1.1 Related Work

Our work in this paper resembles [41] but differs in that: i) our method is cast in the setting of hybrid automata, and therefore rather than generating a single global controlled invariant, we searches for a family of invariants that refine the domain of each mode of the original hybrid automata; ii) a sound and complete criterion is used in invariant generation; iii) various techniques are applied for scalability.

The SOS relaxation approach has been successfully used in safety verification of hybrid systems. In [31,32], the authors used the SOSTOOLS software package [33] to compute *barrier certificates* for polynomial hybrid systems. In [20,48], the authors proposed a hybrid symbolic-numeric approach to compute exact inequality invariants of hybrid systems, by first solving (bilinear) SOS programming numerically and then applying *rational vector recovery* techniques.

A necessary and sufficient condition for positive invariance of convex polyhedra for linear continuous systems was provided in [7]. This condition is extended to linear systems with open polyhedral domain for our need in this paper. Template polyhedra were used in [36,35] to compute positive invariants of hybrid systems by *policy iteration*, which differs from our treatment of the problem using BMI; besides, unlike [35], we do not require the polyhedral invariant to be

generated to have the same shape as the domain. Recently, a method for computing polytopic invariants for polynomial dynamical systems using template polyhedra and linear programming was proposed [38].

Mathematical programming techniques and relevant numerical solvers have also been widely applied to static program analysis. Actually, the template polyhedra abstract domain was first proposed in [37] to generate linear program invariants using linear programming. In [8], to verify invariance and termination of semi-algebraic programs, verification conditions are abstracted into numerical constraints using Lagrangian relaxation or SOS relaxation, which are then resolved by efficient SDP solvers.

In our recent work [49], we studied an optimal switching controller synthesis problem arising from an industrial oil pump system with piece-wise constant continuous dynamics. A hybrid approach combining symbolic computation with numerical computation was developed to synthesize safe controllers with better optimal values.

2 Hybrid Systems and Switching Controller Synthesis Problem

We use hybrid automata to model hybrid systems.

Definition 1 (Hybrid Automaton). *A hybrid automaton (HA) is a system* $\mathcal{H} \widehat{=} (Q, X, f, D, E, G)$*, where*

- $Q = \{q_1, \ldots, q_m\}$ *is a finite set of modes;*
- $X = \{x_1, \ldots, x_n\}$ *is a finite set of continuous state variables, with* $\mathbf{x} = (x_1, \ldots, x_n)$ *ranging over* $\mathbb{R}^n$*;* $Q \times \mathbb{R}^n$ *is the state space of* $\mathcal{H}$*;*
- $f : Q \to (\mathbb{R}^n \to \mathbb{R}^n)$ *assigns to each mode* $q \in Q$ *a vector field* $\mathbf{f}_q$*;*
- $D : Q \to 2^{\mathbb{R}^n}$ *assigns to each mode* $q \in Q$ *a domain* $D_q \subseteq \mathbb{R}^n$*;*
- $E \subseteq Q \times Q$ *is a set of discrete transitions;*
- $G : E \to 2^{\mathbb{R}^n}$ *assigns to each transition* $e \in E$ *a switching guard* $G_e \subseteq \mathbb{R}^n$ *.*

Compared with the conventional versions of HA as in [2], in Definition 1 we make the following assumptions:

- for all $q \in Q$, $\mathbf{f}_q$ is a *polynomial* vector function, and thus the existence and uniqueness of solutions to $\dot{\mathbf{x}} = \mathbf{f}_q$ is guaranteed; besides, $\mathbf{f}_q$ is required to be a *complete*[2] vector field, that is, for any $\mathbf{x}_0 \in \mathbb{R}^n$, the solution $\mathbf{x}(t)$ to $\dot{\mathbf{x}} = \mathbf{f}_q$ exists for all $t \in [0, \infty)$; however, unlike [17], no assumption is made about whether a closed form solution to $\dot{\mathbf{x}} = \mathbf{f}_q$ exists;
- for all $q \in Q$ and all $e \in E$, D_q and G_e are *closed* semi-algebraic sets;
- the initial set of each mode is identical with the domain, and thus omitted;
- all resets are assumed to be identity mappings for ease of presentation, but can also be generalized to polynomial functions.

[2] This assumption is used in the proof of Theorem 1, to exclude the possibility that a hybrid system is blocked due to the inextensibility of trajectories defined by $\mathbf{f}$.

We use a nuclear reactor system discussed in [3,12,17] as a running example throughout this paper.

Example 1. The nuclear reactor system consists of a reactor core and a cooling rod which is immersed into and removed out of the core periodically to keep the temperature of the core, denoted by x, in a certain range. Denote the fraction of the rod immersed into the reactor by θ. Then the initial specification of this system can be represented using the hybrid automaton in Fig. 1.

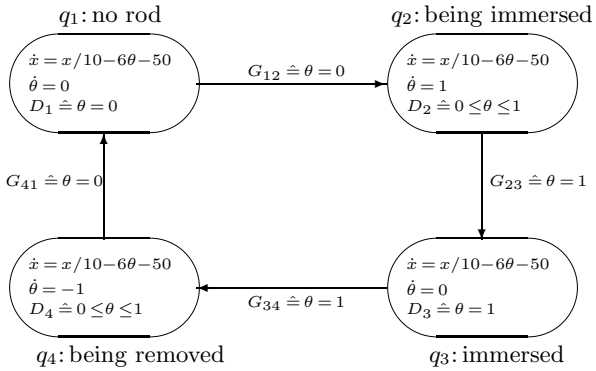


Fig. 1. Nuclear reactor temperature control

The semantics of a hybrid automaton $\mathcal{H}$ can be defined by the set of trajectories it accepts. For the formal definitions of *hybrid time set* and *hybrid trajectory* the readers are referred to [45].

The domain of a hybrid automaton $\mathcal{H}$ is defined as $D_{\mathcal{H}} \widehat{=} \bigcup_{q \in Q}(\{q\} \times D_q)$. We call $\mathcal{H}$ *non-blocking* if for any $(q, \mathbf{x}) \in D_{\mathcal{H}}$, there is a hybrid trajectory from $(q, \mathbf{x})$ which can either be extended to infinite time $t = \infty$ or execute infinitely many discrete transitions; otherwise $\mathcal{H}$ is called *blocking*.

A *safety requirement* $\mathcal{S}$ assigns to each mode $q \in Q$ a safe region $S_q \subseteq \mathbb{R}^n$, i.e. $\mathcal{S} = \bigcup_{q \in Q}(\{q\} \times S_q)$. Alternatively, there could be a global safe region S which all modes are required to satisfy, i.e. $S_q = S$ for all $q \in Q$.

One way of formulating a switching controller synthesis problem for meeting a safety requirement can be precisely defined as follows [4].

Problem 1 (Controller Synthesis for Safety). Given a hybrid automaton $\mathcal{H}$ and a safety property S, find a hybrid automaton $\mathcal{H}' = (Q, X, f, D', E, G')$ such that

(r1) Refinement: for any $q \in Q$, $D'_q \subseteq D_q$, and for any $e \in E$, $G'_e \subseteq G_e$;
(r2) Safety: for any trajectory ω that $\mathcal{H}'$ accepts, if $(q, \mathbf{x})$ is on ω, then $\mathbf{x} \in S_q$;
(r3) Non-blocking: $\mathcal{H}'$ is non-blocking.

If such $\mathcal{H}'$ exists, then $\mathcal{SC} \widehat{=} \{G'_e \subseteq \mathbb{R}^n \mid e \in E\}$ is a safe *switching controller* associated with the set of transitions E, and $D_{\mathcal{H}'} \widehat{=} \bigcup_{q \in Q}(\{q\} \times D'_q)$ is the *controlled invariant set* rendered by $\mathcal{SC}$.

Informally, the switching controller synthesis problem reduces to identifying a set of continuous states for each transition, only at which the system is allowed

to switch from one mode to another, guaranteeing that the system satisfies the safety requirements imposed on every mode as well as can run forever.

3 A QE-Based Approach

In this section, we propose a quantifier elimination (QE) based approach for synthesizing a switching controller for a hybrid automaton by integrating heuristics based on qualitative analysis [17] for predefining templates of invariants, into a relatively complete method for generating semi-algebraic invariants for polynomial continuous dynamical systems with domain [22]. Below we first review the concept of invariant used in [22] based on a related concept in [28].

3.1 Invariants for Continuous Dynamical System with Domain

The notion of *positively invariant set* plays a very important role in the study of continuous dynamical systems [5].

Definition 2. *A subset $P \subseteq \mathbb{R}^n$ is called a positively invariant set for a system $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$, if for all $\mathbf{x}_0 \in P$, the solution $\mathbf{x}(t)$ to $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ starting from $\mathbf{x}_0$ satisfies $\mathbf{x}(t) \in P$ for all $t > 0$.*

However, the above concept of invariant is not suitable for the study of hybrid systems. The reason is that each mode of a hybrid automaton $\mathcal{H}$ can be abstracted as a pair $(D, \mathbf{f})$, where D and $\mathbf{f}$ are the domain and vector field of a certain mode of $\mathcal{H}$; for any trajectory $\mathbf{x}(t)$ of $\mathbf{f}$, only the part of $\mathbf{x}(t)$ that lies in D is meaningful to the behavior of $\mathcal{H}$, rather than the complete trajectory with all $t > 0$ as in Definition 2. Therefore the following concept of invariant is proposed for systems like $(D, \mathbf{f})$.

Definition 3. *A subset $P \subseteq \mathbb{R}^n$ is called an invariant of $(D, \mathbf{f})$, if for all $\mathbf{x}_0 \in P$ and all $T \geq 0$, the solution $\mathbf{x}(t)$ of $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ over $[0, T]$ with $\mathbf{x}(0) = \mathbf{x}_0$ satisfies*

$$(\forall t \in [0, T].\, \mathbf{x}(t) \in D) \longrightarrow (\forall t \in [0, T].\, \mathbf{x}(t) \in P) \ .$$

Intuitively, P is an invariant of $(D, \mathbf{f})$ if any continuous evolution starting from P stays in P as long as it stays in D. If $D = \mathbb{R}^n$, then an invariant of $(D, \mathbf{f})$ is a positively invariant set of $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ as defined in Definition 2; otherwise if D is a proper subset of $\mathbb{R}^n$, then generally the notion of invariant in Definition 3 is weaker, and thus allows a broader class of sets to be invariants.

Example 2. Suppose $D \mathrel{\widehat{=}} x > 0$ and $\mathbf{f} = (-y, x)$. It can be shown (please to the full version [18]) that $P \mathrel{\widehat{=}} y \geq 0$ is not a positively invariant set of $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$, but is an invariant of $(D, \mathbf{f})$.

The above arguments show that when dealing with continuous evolutions in the context of hybrid system, it is necessary to study invariants for augmented systems $(D, \mathbf{f})$, rather than pure continuous dynamical systems $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$.

3.2 The Abstract Synthesis Procedure

Solving Problem 1 amounts to refining the domains and guards of $\mathcal{H}$ by removing so-called *bad* states. A state $(q, \mathbf{x}) \in D_{\mathcal{H}}$ is *bad* if the hybrid trajectory starting from $(q, \mathbf{x})$ either blocks $\mathcal{H}$ or violates $\mathcal{S}$; otherwise, it is called a *good* state. From Definition 3, we observe that the set of good states of $\mathcal{H}$ can be approximated using invariants, which results in the following solution to Problem 1.

Theorem 1. *Let $\mathcal{H}$ and $\mathcal{S}$ be as in Problem 1. Further, for each $q \in Q$, let D'_q be a closed subset of $\mathbb{R}^n$ such that $\bigcup_{q\in Q} D'_q$ is non-empty (to imply at least one D'_q is non-empty). If we have*

(c1) for all $q \in Q$, $D'_q \subseteq D_q \cap S_q$;
(c2) for all $q \in Q$, D'_q is an invariant of $(H_q, \mathbf{f}_q)$ with

$$H_q \mathrel{\widehat{=}} \Big(\bigcup_{e=(q,q')\in E} G'_e \Big)^c,$$

where $G'_e \mathrel{\widehat{=}} G_e \cap D'_{q'}$ for $e = (q, q')$, and A^c denotes the complement of A in $\mathbb{R}^n$, then the HA $\mathcal{H}' = (Q, X, f, D', E, G')$ is a solution to Problem 1.

Proof. Please refer to the full version of this paper [18]. □

In Theorem 1, condition (c1) ensures that D'_q is a refinement of D_q and mode q satisfies its safety condition, thus guaranteeing (r1) and (r2) of Problem 1; condition (c2) requires that any trajectory starting in mode q will either remain in mode q or jump to another mode q' when the associated guard is satisfied, thus guaranteeing (r3) of Problem 1.

Based on Theorem 1, we give below the steps of a template-based method for synthesizing a switching controller.

(s1) **Template assignment:** assign to each $q \in Q$ a template parametrically specifying D'_q, which will be required (see step (s3)) to be a refinement of D_q, as well as the invariant to be generated at mode q;
(s2) **Guard refinement:** refine guard G_e for each $e = (q, q') \in E$ by setting $G'_e \mathrel{\widehat{=}} G_e \cap D'_{q'}$;
(s3) **Deriving synthesis conditions:** encode (c1) and (c2) in Theorem 1 into constraints on parameters appearing in the templates;
(s4) **Constraint solving:** solve the constraints derived from (s3) in terms of the parameters;
(s5) **Parameters instantiation:** find an appropriate instantiation of D'_q and G'_e such that D'_q are closed[3] sets for all $q \in Q$, and D'_q is non-empty[4] for at least one $q \in Q$; if such an instantiation is not found, we choose a new set of templates and go back to (s1).

[3] This can be enforced by restricting to $\geq, \leq, =$ and $\vee, \wedge$ symbols in templates.
[4] To avoid trivially generating an empty set, some additional constraints can be encoded in step (s3).

We have assumed the hybrid automata to be specified by polynomial expressions. If in addition we restrict the form of safety requirements and templates to polynomial formulas, then computability of the above abstract procedure is guaranteed by Tarski's result [44].

In (s3), condition (c1) can be encoded into a first-order polynomial formula straightforwardly; encoding of (c2) into first-order polynomial constraints is based on our previous work in [22] on a relatively complete method for generating invariants (see Section 3.3). We use *quantifier elimination* (QE) to solve the first-order polynomial constraints obtained in (s4).

The shape of chosen templates in (s1) determines the likelihood of success of the above procedure, as well as the complexity of QE in (s4). In Section 3.5, we discuss heuristics for choosing appropriate templates using the *qualitative analysis* discussed in [17].

3.3 A Relatively Complete Method for Generating Invariants

In [22] we presented a sound and relatively complete approach for generating *semi-algebraic* invariants for $(D, \mathbf{f})$ with semi-algebraic domain D and polynomial vector function $\mathbf{f}$. For self-containedness, we introduce below the main result in the simplest case. For details the readers can consult [22].

Given a polynomial $p(\mathbf{x})$ and a polynomial vector field $\mathbf{f}(\mathbf{x})$, define the *Lie derivatives* of p along $\mathbf{f}$, $L^k_{\mathbf{f}}p : \mathbb{R}^n \longrightarrow \mathbb{R}$ for $k \in \mathbb{N}$, as follows:

- $L^0_{\mathbf{f}}p(\mathbf{x}) = p(\mathbf{x})$;
- $L^k_{\mathbf{f}}p(\mathbf{x}) = \langle \nabla L^{k-1}_{\mathbf{f}}p(\mathbf{x}), \mathbf{f}(\mathbf{x}) \rangle$, for $n > 0$,

where $\nabla g(\mathbf{x})$ denotes the *gradient* vector of a scalar function $g(\mathbf{x})$, and $\langle \cdot, \cdot \rangle$ is the *inner product* of two vectors.

Example 3. Suppose $\mathbf{f} = (1, 1)$ and $p(x, y) = -x^2 + y$. Then $L^0_{\mathbf{f}}p(x, y) = -x^2 + y$, $L^1_{\mathbf{f}}p(x, y) = -2x + 1$, $L^2_{\mathbf{f}}p(x, y) = -2$, and $L^k_{\mathbf{f}}p(x, y) = 0$ for all $k \geq 3$.

The importance of Lie derivatives is that they can be used to predict continuous evolutions of $\mathbf{f}$ in terms of a polynomial p. To illustrate this, look at Fig. 2 showing the vector field $\mathbf{f}$ (small arrows) and the semi-algebraic set $P \mathrel{\widehat{=}} p \geq 0$ (grey area), with $\mathbf{f}$ and p defined in Example 3. At point $A_0(-1, 1)$ on the boundary of P, the first-order Lie derivative $L^1_{\mathbf{f}}p(-1, 1) = 3 > 0$, indicating that the angle between the vector field $(1, 1)$ (arrow $\overrightarrow{A_0A_1}$), and the gradient $\nabla p(-1, 1) = (2, 1)$ (arrow $\overrightarrow{A_0A_2}$) is less that $\frac{\pi}{2}$, which further indicates that the trajectory of $\mathbf{f}$ from A_0 (arrow $\overrightarrow{A_0A_3}$) would move towards the $p > 0$ side.

At point $B_0(\frac{1}{2}, \frac{1}{4})$, $L^1_{\mathbf{f}}p(\frac{1}{2}, \frac{1}{4}) = 0$ indicates that the vector field $\overrightarrow{B_0B_1}$ is orthogonal to the gradient $\overrightarrow{B_0B_2}$, from which we cannot tell how the trajectory from B_0 (arrow $\overrightarrow{B_0B_3}$) evolves with respect to P. However, if we resort to higher order Lie derivatives, then from $L^2_{\mathbf{f}}p(\frac{1}{2}, \frac{1}{4}) = -2 < 0$ we assert that $\overrightarrow{B_0B_3}$ would go out of P into the $p < 0$ side immediately.

Generally, given p and $\mathbf{f}$, to make predictions as above at a point $\mathbf{x} \in \mathbb{R}^n$, we need to compute $L^0_{\mathbf{f}}p(\mathbf{x}), L^1_{\mathbf{f}}p(\mathbf{x}), \ldots$ to get the first $k \in \mathbb{N}$ s.t. $L^k_{\mathbf{f}}p(\mathbf{x}) \neq 0$.

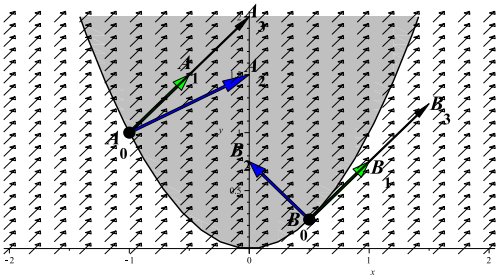


Fig. 2. Using Lie derivatives to predict continuous evolution

Furthermore, we can compute an integer $N_{p,\mathbf{f}}$ from p and $\mathbf{f}$ such that if all Lie derivatives with order $\leq N_{p,\mathbf{f}}$ evaluate to 0 at $\mathbf{x}$, then $L^k_{\mathbf{f}}(\mathbf{x}) = 0$ for all $k \in \mathbb{N}$. As a result, it suffices to compute Lie derivatives up to the $N_{p,\mathbf{f}}$-th order.

Formally, we have

Theorem 2. *Given a system $(D, \mathbf{f})$ with $D \mathrel{\widehat{=}} h(\mathbf{x}) > 0$, it has an invariant of the form $P \mathrel{\widehat{=}} p(\mathbf{x}) \geq 0$ if and only if $\forall \mathbf{x}.\big(p(\mathbf{x}) = 0 \wedge h(\mathbf{x}) > 0 \longrightarrow \psi(p, \mathbf{f})\big)$, where*

$$\psi(p,\mathbf{f}) \mathrel{\widehat{=}} \begin{array}{ll} & L^1_{\mathbf{f}}p(\mathbf{x}) > 0 \\ \vee & L^1_{\mathbf{f}}p(\mathbf{x}) = 0 \wedge L^2_{\mathbf{f}}p(\mathbf{x}) > 0 \\ \vee & \cdots \\ \vee & L^1_{\mathbf{f}}p(\mathbf{x}) = 0 \wedge \cdots \wedge L^{N_{p,\mathbf{f}}-1}_{\mathbf{f}}p(\mathbf{x}) = 0 \wedge L^{N_{p,\mathbf{f}}}_{\mathbf{f}}p(\mathbf{x}) > 0 \\ \vee & L^1_{\mathbf{f}}p(\mathbf{x}) = 0 \wedge \cdots \wedge L^{N_{p,\mathbf{f}}-1}_{\mathbf{f}}p(\mathbf{x}) = 0 \wedge L^{N_{p,\mathbf{f}}}_{\mathbf{f}}p(\mathbf{x}) = 0 \end{array}.$$

Proof. Please refer to [22]. □

The above theorem can be generalized for parametric polynomials $p(\mathbf{u}, \mathbf{x})$, thus enabling us to use polynomial templates and QE to automatically discover invariants. Such a method for invariant generation is relatively complete, that is, if there exists invariants in the form of the predefined template, then we are able to find one.

3.4 Comparison with Other Invariant Generation Approaches

In Platzer et al's work [28,27,30,29] and Tiwari et al's work [11,42,41], various criteria are proposed for checking invariants for systems $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ or $(D, \mathbf{f})$. We will show the strength of our criterion due to its completeness through the following comparison.

Consider the system $(\mathbb{R}^2, \mathbf{f})$ from [42] with $\mathbf{f} \mathrel{\widehat{=}} (1-y, x)$. It can be shown that this system has an invariant $p \geq 0$ with $p \mathrel{\widehat{=}} -(-x^2 - y^2 + 2y)^2$. Geometrically, the trajectories of $\mathbf{f}$ are all circles centered at $(0, 1)$. The set $p \geq 0$, or equivalently $p = 0$ is actually one of these circles with radius 1, thus an invariant.

In [42], sound and complete inference rules are given for invariants that are linear, quadratic, smooth or convex. However, it was also pointed out in [42] that all these rules failed to prove the invariance property of $p \geq 0$, for in this example $p \geq 0$ is not linear or quadratic, nor is it smooth or convex. Furthermore, by a simple computation we get $L_{\mathbf{f}}^k p \equiv 0$ for all $k \geq 1$, so the sound but incomplete rule in [42,41] which involves only strict inequalities over finite-order Lie derivatives is also inapplicable. However, from $L_{\mathbf{f}}^1 p \equiv 0$ we get[5] $N_{p,\mathbf{f}} = 0$, and then according to Theorem 2, $p \geq 0$ can be verified since $\forall x \forall y.\ \big(-(-x^2 - y^2 + 2y)^2 = 0 \longrightarrow \mathsf{true}\big)$ holds trivially.

Although the rule in [28] can also be used to check the invariant $p \geq 0$, generally it only works on very restricted invariants. Even for linear systems like $(\mathbb{R}, \dot{x} = x)$, it cannot prove the invariant $x \geq 0$ because the verification condition $\forall x.x \geq 0$ is obviously *false*; whereas our approach requires $\forall x.(x = 0 \rightarrow \mathsf{true})$ (for this example $N_{p,\mathbf{f}}$=0 so $\psi(p, \mathbf{f})$ in Theorem 2 is true), which is trivially *true*.

Intuitively, to prove that $p \geq 0$ is an invariant of $(D, \mathbf{f})$, the rule in [28] requires $L_{\mathbf{f}}^1 p \geq 0$ over the whole domain D, while the rule in [42,41] requires that on the boundary of $p \geq 0$ inside D, the first non-zero high order Lie derivative, say $L_{\mathbf{f}}^k p$, is strictly positive. Completeness is lost either because non-boundary points are unnecessarily examined, or an upper bound on the order of Lie derivatives to be considered (the number $N_{p,\mathbf{f}}$ in our rule) is not given.

The above analysis shows the generality of our approach, using which it is possible to generate invariants in many general cases, and hence gives more possibility to synthesize a controller based on our understandings of the kind of controllers that can be synthesized using methods in [41,43,40].

3.5 Heuristics for Predefining Templates

The key steps of the qualitative analysis used in [17] are as follows.

1. Infer the evolution behavior (increasing or decreasing) of continuous variables in each mode from the differential equations (using first or second order derivatives).
2. Identify modes at which the evolution behavior (increasing or decreasing) of a continuous variable changes, and thus the maximal (or minimal) value of this continuous variable can be achieved. Such modes are called *control critical* modes.
3. At a control critical mode, equate the maximal (or minimal) value of a continuous variable to the corresponding safety upper (or lower) bound to obtain a specific continuous state, called a *critical point*.
4. At a control critical mode, use the critical point as the initial value to compute a closed form solution of the differential equation at that mode; then backtrack along this solution to compute a switching point which evokes a transition leading to the control critical mode.
5. The above obtained switching point is chosen as a new critical point, which is then backward propagated to other modes in a similar way.

[5] How to compute $N_{p,\mathbf{f}}$ for polynomial functions p and $\mathbf{f}$ can be found in [22].

Next, we illustrate how such an analysis helps in predefining templates for the running example.

Example 4 (Nuclear Reactor Temperature Control). Our goal is to synthesize a switching controller for the system in Example 1 with the global safety requirement that the temperature of the core lies between 510 and 550, i.e. $S_i \mathrel{\widehat{=}} 510 \leq x \leq 550$ for $i = 1, 2, 3, 4$. Please refer to Fig. 3 to get a better understanding of following discussions.

1) **Refine domains.** Using the safety requirement, domains D_i for $i = 1, 2, 3, 4$ are refined by $D_i^s \mathrel{\widehat{=}} D_i \cap S_i$, e.g. $D_1^s \mathrel{\widehat{=}} \theta = 0 \wedge 510 \leq x \leq 550$.
2) **Infer continuous evolutions.** Let $l_1 \mathrel{\widehat{=}} x/10 - 6\theta - 50 = 0$ be the *zero-level* set of $\dot{x}$ and check how x and θ evolve in each mode. For example, in D_2^s, $\dot{x} > 0$ on the left of l_1 and $\dot{x} < 0$ on the right; since θ increases from 0 to 1, x first increases then decreases and achieves maximal value when crossing l_1.
3) **Identify control critical modes.** By 2), q_2 and q_4 are control critical modes at which the evolution direction of x changes and the maximal (or minimal) value of x is achieved.
4) **Generate critical points.** By 3), we can get a critical point $E(5/6, 550)$ at q_2 by taking the intersection of l_1 and the safety upper bound $x = 550$; and $F(1/6, 510)$ can be obtained similarly at q_4.
5) **Propagate critical points.** E is backward propagated to $A(0, a)$ using the trajectory $\widehat{AE}$ through E defined by $\mathbf{f}_{q_2}$, and then to $C(1, c)$ using the trajectory $\widehat{CA}$ through A defined by $\mathbf{f}_{q_4}$; similarly, by propagating F we get D and B.
6) **Construct templates.** For brevity, we only show how to construct D_2'. Intuitively, $\theta = 0$, $\theta = 1$, $\widehat{AE}$ and $\widehat{BD}$ form the boundaries of D_2'. In order to get a semi-algebraic template, we need to fit $\widehat{AE}$ and $\widehat{BD}$ (which are generally not polynomial curves) by polynomials using points A, E and B, D respectively. By the inference of 2), $\widehat{AE}$ has only one extreme point (also the maximum point) E in D_2^s, and is tangential to $x = 550$ at E. A simple algebraic curve that can exhibit a shape similar to $\widehat{AE}$ is the parabola through A, E opening downward with $l_2 \mathrel{\widehat{=}} \theta = \frac{5}{6}$ the axis of symmetry. Therefore to minimize the degree of terms appearing in templates, we do not resort to polynomials with degree greater than 2. This parabola can be computed using the coordinates of A, E as: $x - 550 - \frac{36}{25}(a - 550)(\theta - \frac{5}{6})^2 = 0$, with a the parameter to be determined.

Through the above analysis, we generate the following templates:

- $D_1' \mathrel{\widehat{=}} \theta = 0 \wedge 510 \leq x \leq a$;
- $D_2' \mathrel{\widehat{=}} 0 \leq \theta \leq 1 \wedge x - b \geq \theta(d - b) \wedge x - 550 - \frac{36}{25}(a - 550)(\theta - \frac{5}{6})^2 \leq 0$;
- $D_3' \mathrel{\widehat{=}} \theta = 1 \wedge d \leq x \leq 550$;
- $D_4' \mathrel{\widehat{=}} 0 \leq \theta \leq 1 \wedge x - a \leq \theta(c - a) \wedge x - 510 - \frac{36}{25}(d - 510)(\theta - \frac{1}{6})^2 \geq 0$,

in which a, b, c, d are parameters satisfying

$$510 \leq b \leq a \leq 550 \wedge 510 \leq d \leq c \leq 550\,.$$

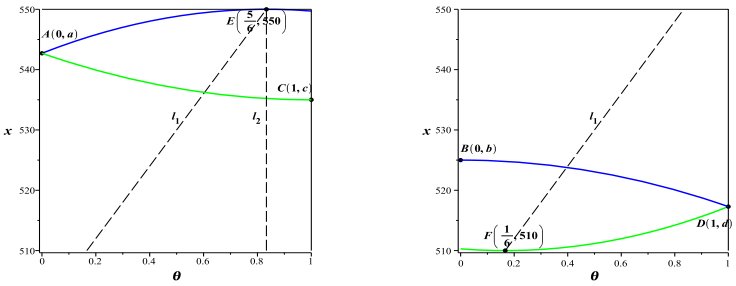


Fig. 3. Predefining templates via qualitative analysis

Note that without qualitative analysis, a single generic *quadratic* polynomial over θ and x would require $\binom{2+2}{2} = 6$ parameters.

Based on the framework presented in Section 3.2, we show below how to synthesize a switching controller for the system in Example 4 step by step.

Example 5 (Nuclear Reactor Temperature Control Contd.).

(s1) The four templates are defined in Example 4.

(s2) The four guards are refined by $G'_{ij} \mathrel{\widehat{=}} G_{ij} \cap D'_j$ and then simplified to:
- $G'_{12} \mathrel{\widehat{=}} \theta = 0 \wedge b \leq x \leq a$;
- $G'_{23} \mathrel{\widehat{=}} \theta = 1 \wedge d \leq x \leq 550$;
- $G'_{34} \mathrel{\widehat{=}} \theta = 1 \wedge d \leq x \leq c$;
- $G'_{41} \mathrel{\widehat{=}} \theta = 0 \wedge 510 \leq x \leq a$.

(s3) Based on Theorem 1 and a general version of Theorem 2 [22], we can derive the synthesis condition, which is a first-order polynomial formula in the form of $\phi \mathrel{\widehat{=}} \forall x \forall \theta . \varphi(a, b, c, d, x, \theta)$. We do not include ϕ here due to its big size.

(s4) By applying QE to ϕ, we get the following solution to the parameters:[6]

$$a = \frac{6575}{12} \wedge b = \frac{4135}{8} \wedge c = \frac{4345}{8} \wedge d = \frac{6145}{12} \ . \tag{1}$$

(s5) Instantiate D'_i and G'_{ij} by (1). It is obvious that all D'_i are nonempty closed sets. According to Theorem 1, we get a safe switching controller for the nuclear reactor system, i.e.
- $G'_{12} \mathrel{\widehat{=}} \theta = 0 \wedge 4135/8 \leq x \leq 6575/12$;
- $G'_{23} \mathrel{\widehat{=}} \theta = 1 \wedge 6145/12 \leq x \leq 550$;
- $G'_{34} \mathrel{\widehat{=}} \theta = 1 \wedge 6145/12 \leq x \leq 4345/8$;
- $G'_{41} \mathrel{\widehat{=}} \theta = 0 \wedge 510 \leq x \leq 6575/12$.

In [17], an upper bound $x = 547.97$ for G_{12} and a lower bound $x = 512.03$ for G_{34} are obtained by solving the differential equations at mode q_2 and q_4 respectively.

[6] The process of applying QE and selecting a sample solution demands some human effort which can be found in the full version of this paper [18].

By (1), the corresponding bounds generated here are $x \leq \frac{6575}{12} = 547.92$ and $x \geq \frac{6145}{12} = 512.08$.

As should be evident from the above results, in contrast to [17], where differential equations are solved to get closed-form solutions, we are able to get good results without requiring closed-form solutions. This indicates that our approach should work well for hybrid automata where differential equations for modes need not have closed form solutions.

4 Speeding Computations Using Numerical Methods on Specialized Templates

The QE-based approach crucially depends upon quantifier elimination techniques. It is well known that the complexity of a general purpose QE method over the full theory of real-closed fields is *doubly exponential* in the number of variables [9]. Therefore it is desirable to develop heuristics to do QE more efficiently. As shown in Section 3.5, qualitative analysis helps in reducing the number of parameters in templates. Another possible way to address the issue of high computational cost is resorting to numerical methods. In this section, we will discuss the application of two such approaches on specialized templates.

4.1 The SOS Relaxation Approach

Let $\mathbb{R}[x_1, x_2, \ldots, x_n]$, or $\mathbb{R}[\mathbf{x}]$ for short, denote the polynomial ring over variables $x_1, x_2, \ldots, x_n$ with real coefficients. A *monomial* is an expression in the form of $x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n}$ with $(\alpha_1, \alpha_2, \ldots, \alpha_n) \in \mathbb{N}^n$. A *polynomial* $p(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$ of degree d can be written as a linear combination of $\binom{n+d}{d}$ monomials, i.e.

$$p(\mathbf{x}) = \sum_{\alpha_1+\alpha_2+\cdots+\alpha_n \leq d} c_{(\alpha_1,\alpha_2,\ldots,\alpha_n)} \cdot x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n} \quad .$$

We call p an SOS (sum-of-squares) if there exist s polynomials $q_1, q_2, \ldots, q_s$ s.t.

$$p = \sum_{1 \leq i \leq s} q_i^2 \quad .$$

It is obvious that any SOS p is non-negative, i.e. $\forall \mathbf{x} \in \mathbb{R}^n . p(\mathbf{x}) \geq 0$.

The basic idea of SOS relaxation is as follows: to prove that a polynomial p is nonnegative, it suffices to show that p can be decomposed into a sum of squares, a trivially sufficient condition for non-negativity (but generally not necessary); similarly, to prove $p \geq 0$ on the semi-algebraic set $q \geq 0$, it is sufficient to find two SOS r_1, r_2 such that $p = r_1 + r_2 \cdot q$.

SOS relaxation is attractive because SOS decomposition can be reduced to a semi-definite programming (SDP) problem according to the following equivalence [26]:

> A polynomial p of degree $2d$ is an SOS if and only if there exists a semi-definite matrix Q such that $p = \mathbf{q} \cdot Q \cdot \mathbf{q}^T$, where $\mathbf{q}$ is a $\binom{n+d}{d}$-dimensional row vector of monomials with degree $\leq d$.

SDP is a convex programming that is solvable in *polynomial* time using numerical methods such as the interior point method [47]. Therefore the searching for SOS is a tractable problem.

We now show how SOS can be related to invariant generation. Let $p \geq 0$ be a parametric template defined for the system $(h > 0, \mathbf{f})$. By Theorem 2, a sufficient condition for $p \geq 0$ to be an invariant of $(h > 0, \mathbf{f})$ is

$$\forall \mathbf{x}. \Big(p(\mathbf{x}) = 0 \wedge h(\mathbf{x}) > 0 \longrightarrow L^1_{\mathbf{f}} p(\mathbf{x}) > 0 \Big) ,$$

of which a sufficient condition is

$$\forall \mathbf{x}. \Big(h(\mathbf{x}) > 0 \longrightarrow L^1_{\mathbf{f}} p(\mathbf{x}) > 0 \Big) ,$$

and again of which a sufficient condition given by SOS relaxation is

$$L^1_{\mathbf{f}} p = s_1 + s_2 \cdot h + \varepsilon \ , \tag{2}$$

where s_1, s_2 are SOS and ε is a positive constant. By expressing s_1, s_2 via unknown semi-definite matrices and equating the parametric coefficients of monomials on both sides of (2), we can obtain an SDP problem, the solution of which gives an invariant $p \geq 0$.

As shown above, in general, a constraint that possesses easy SOS relaxation encoding has the form $\forall \mathbf{x}.(\bigwedge_{i=1}^{m} g_i \triangleright 0 \longrightarrow g_{m+1} \triangleright 0)$, where all g_i's are polynomials in $\mathbf{x}$ and $\triangleright \in \{\geq, >\}$. Handling arbitrary Boolean combinations, which is common case in Theorem 1 and 2, is not the strength of the SOS approach. For the particular purpose of facilitating SOS encodings of controller synthesis conditions, we propose specialized use of both Theorem 1 and 2, which can be found in the full version of this paper [18].

For the nuclear reactor example, we define two general *quartic* templates in the form of

$$\theta \geq 0 \wedge \theta \leq 1 \wedge \sum_{\alpha_1 + \alpha_2 \leq 4} c_{(\alpha_1, \alpha_2)} \cdot \theta^{\alpha_1} x^{\alpha_2} \leq 0$$

for mode q_2 and q_4. The idea is to fit the two boundaries of D'_2 (or D'_4), i.e. $\widehat{AE}$ and $\widehat{BD}$ in Fig. 3 simultaneously by one quartic polynomial, rather than two polynomials with lower degrees. The high efficiency of SOS solvers gives such possibility of using generic templates with higher degrees. Using the SOS relaxation techniques discussed above, the following switching controller is obtained:

- $G'_{12} \mathrel{\widehat{=}} \theta = 0 \wedge 519.10 \leq x \leq 547.86$;
- $G'_{23} \mathrel{\widehat{=}} \theta = 1 \wedge 512.09 \leq x \leq 550.00$;
- $G'_{34} \mathrel{\widehat{=}} \theta = 1 \wedge 512.09 \leq x \leq 546.15$;
- $G'_{41} \mathrel{\widehat{=}} \theta = 0 \wedge 510.00 \leq x \leq 547.86$.

For more details on the issues of defining templates, encoding constraints, applying numerical solvers etc, please refer to the full version [18].

4.2 The Template Polyhedra Approach

Convex polyhedra are a popular class of (positive) invariant sets of linear (continuous or discrete) systems [5]. A *convex polyhedron* in $\mathbb{R}^n$ can be represented using linear inequality constraints as $Q\mathbf{x} \leq \rho$, where $Q \in \mathbb{R}^{r\times n}$ is an $r \times n$ matrix, and $\mathbf{x} \in \mathbb{R}^{n\times 1}, \rho \in \mathbb{R}^{r\times 1}$ are column vectors.

Given a linear continuous dynamical system $\dot{\mathbf{x}} = A\mathbf{x}$ with $A \in \mathbb{R}^{n\times n}$, the following result on (positive) polyhedral invariant set is established in [7].

Proposition 1. *The polyhedron $Q\mathbf{x} \leq \rho$ is a positive invariant set of $\dot{\mathbf{x}} = A\mathbf{x}$ if and only if there exists an essentially non-negative*[7] *matrix $H \in \mathbb{R}^{r\times r}$ satisfying $HQ = QA$ and $H\rho \leq 0$.*

By simply applying the famous *Farkas' lemma* [11], we can generalize Proposition 1 and give a sufficient condition for polyhedral invariants of linear dynamics with *open* polyhedral domain (below we use a simple domain for ease of presentation).

Proposition 2. *Let $\mathbf{f} \mathrel{\widehat{=}} A\mathbf{x} + \mathbf{b}$ and $D \mathrel{\widehat{=}} \mathbf{cx} < a$, where $a \in \mathbb{R}$, $\mathbf{b} \in \mathbb{R}^{n\times 1}$ is a column vector, and $\mathbf{c} \in \mathbb{R}^{1\times n}$ is a row vector. Then the polyhedron $Q\mathbf{x} \leq \rho$ is an invariant of the system $(D, \mathbf{f})$, if there exists an essentially non-negative matrix $H \in \mathbb{R}^{r\times r}$ and a non-negative column vector $\lambda \geq 0$ in $\mathbb{R}^{r\times 1}$ s.t.*

(1) $HQ = \mathsf{diag}(\lambda)QA - \mathsf{ones}^{(r,1)}\mathbf{c}$ *; and*
(2) $H\rho \leq -\mathsf{diag}(\lambda)Q\mathbf{b} - \mathsf{ones}^{(r,1)}a$ *,*

where $\mathsf{diag}(\lambda)$ *denotes the $r \times r$ diagonal matrix whose main diagonal corresponds to the vector λ, and* $\mathsf{ones}^{(r,1)}$ *denotes the $r \times 1$ column vector with all entries* 1*.*

Proof. Please refer to the full version of this paper [18]. □

Proposition 2 serves as the basis of automatic generation of polyhedral invariants for linear systems. To reduce the number of parameters in a polyhedral template, we propose the use of *template polyhedra*. The idea is to partly fix the shape of the invariant polyhedra by fixing the orientation of their facets. Formally, a template polyhedron is of the form $Q\mathbf{x} \leq \rho$ where Q is fixed a priori and ρ is to be determined. Any instantiation of ρ from $\mathbb{R}^{r\times 1}$ produces a concrete polyhedron. In this paper, since the system is planar, we choose Q in such a way that its row vectors form a set of uniformly distributed directions on a unit circle, i.e.

$$\mathbf{q}_i = \Big(\cos(\frac{i-1}{r}2\pi), \sin(\frac{i-1}{r}2\pi)\Big)$$

for $1 \leq i \leq r$, where $\mathbf{q}_i$ denotes the i-th row of Q. It is easy to see that $Q\mathbf{x} \leq \rho$ is just a rectangle when $r = 4$, and an octagon when $r = 8$.

To determine ρ, we need to find such matrices H and λ satisfying Proposition 2. Note that since both H and ρ are indeterminate, the constraint (2) in

[7] A square matrix is *essentially non-negative* if all its entries are non-negative except for those on the diagonal. Besides, given a matrix M, in this paper the inequalities or equations $M \geq 0$, $M > 0$, $M = 0$ should be interpreted entry-wise.

Proposition 2 becomes *bilinear*, making the problem NP-hard [46] to solve. It is however still tractable using modern BMI solvers.

Using octagonal templates[8] for mode q_2 and q_4 in the nuclear reactor example, we obtain the following switching controller:

- $G'_{12} \mathrel{\widehat{=}} \theta = 0 \wedge 519.90 \leq x \leq 545.70$;
- $G'_{23} \mathrel{\widehat{=}} \theta = 1 \wedge 514.40 \leq x \leq 550.00$;
- $G'_{34} \mathrel{\widehat{=}} \theta = 1 \wedge 514.40 \leq x \leq 538.24$;
- $G'_{41} \mathrel{\widehat{=}} \theta = 0 \wedge 510.00 \leq x \leq 545.70$.

As in Section 4.1, the readers are referred to [18] for more details about the application of the template polyhedra approach.

5 Conclusion and Discussion

We have extended a template-based approach for synthesizing switching controllers for semi-algebraic hybrid systems by combining symbolic invariant generation methods using quantifier elimination with qualitative methods to determine the likely shape of invariants. We have also investigated the application of numerical methods to gain more scalability and automation. A summary comparison of the three proposed approaches, as well as their advantages and problems, are given in the following aspects based on our experience.

- **Applicability.** The QE-based and SOS relaxation approaches can be applied to semi-algebraic systems which do not have closed-form solutions, while the template-polyhedra approach is only applicable to linear systems.
- **Design of Templates.** The QE-based approach demands much heuristics, which currently works well only on systems with low dimension, in determining templates, while the other two require less human effort.
- **Derivation of Synthesis Conditions.** For the QE-based method, derivation of synthesis conditions is a routine work because of the power of first-order formulas in formulating problems; the other two approaches are not good at handling complex logical structures and require the problems to be of specific forms, which usually demands some human work in practice.
- **Quality of Results.** The QE-based and SOS relaxation approaches can generate (non-convex) semi-algebraic invariants, while the template-polyhedra approach can only generate convex polyhedral invariants. Figure 4 demonstrates the synthesized D'_2 for the nuclear reactor example by the three approaches in turn: the first one is formed by straight lines and a parabola, the second one by straight lines and a quartic polynomial curve, and the third one is an octagon. For the switching controller synthesis problem, it's desirable to generate as large as possible invariants to gain more possibility of further refinement of the controllers based on other criteria. In this sense it's difficult to judge the merits of three approaches (e.g. in Fig. 4 the synthesized invariants have similar sizes).

[8] We search for polyhedral invariants using templates with $4, 8, 12, \ldots$ facets, and could not get a solution using templates with 4 facets.

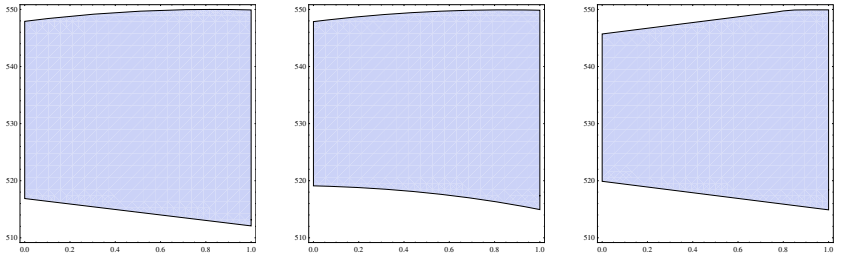


Fig. 4. Shapes of synthesized invariants by three approaches

– **Computational Issues.** For the QE-based approach, we have used the algebraic tool Redlog [10] to perform QE, and run several rounds of QEPCAD [6] (the slfq function) with human interactions to simplify the output of Redlog. For the numerical approaches, we use the MATLAB optimization toolbox YALMIP [24,25] as a high-level modeling environment and the interfaced external solvers SeDuMi [39] and PENBMI [19] (the TOMLAB [13] version) to solve the underlying SDP and BMI problems respectively. In our experiments, the SDP solver exhibits consistently good performances, but the BMI solver frequently runs into numerical problems. To make the BMI solver work we have to adjust the input constraints and the solver options a lot with trials and errors[9]. Table 1 shows the time cost of three approaches

Table 1. Templates and time cost of three controller synthesis approaches

Approach		QE-based	SOS-relaxation	template-polyhedra
Tool		Redlog + slfq	YALMIP + SeDuMi	YALMIP + PENBMI
Template	NR	quadratic, #PARMS = 4	generic quartic	8 facets
	TS	quadratic, #PARMS = 2	generic quartic	12 facets
Time (sec)	NR	12.663	1.969	0.578
	TS	7.092	1.609	1.703

on the nuclear reactor (NR) example as well as a thermostat (TS) example from [14]. All computations are done on a desktop with the Intel Q9400 2.66GHz CPU and 4GB RAM running Ubuntu Linux. We can see that for these two examples the QE-based approach is more expensive in time compared to numerical approaches. However, according to our experience, the template-polyhedra approach does not scale well due to the NP-hardness of BMI problems, so its superiority to QE-based approach may not always be the case. For a detailed explanation of Table 1 as well as the description of the TS example please refer to [18].

[9] Another way is to use the global nonlinear optimization solver BMIBNB in YALMIP, which would cause an increase of time cost by dozens (even hundreds) of times for the same two examples.

– **Soundness.** The QE-based approach is exact while the other two approaches suffer from numerical errors which would cause the synthesis of unsafe controllers. To partly address this problem, we have directly encoded some tolerance of numerical errors into the synthesis conditions to increase robustness and reduce the risk of synthesizing bad controllers. The justification for adopting numerical methods is that verification is much easier than synthesis. For example, for the nuclear reactor example, we have verified posteriorly and symbolically the controllers synthesized by both numerical approaches, and the verification process is very efficient.

Our preliminary analysis suggests the effectiveness of the three proposed approaches. We are currently experimenting with these methods on more examples, especially nonlinear ones which do not possess closed-form solutions. We plan to extend these approaches for reachability and/or optimality requirements as well, by incorporating our previous results on *asymptotic stability* analysis [23] and a case study in optimal control [49].

Acknowledgements. We thank Dr. Jiang Liu for his great contribution to our previous joint work on invariant generation. We also thank Dr. Matthias Horbach, Mr. ThanhVu Nguyen, and the anonymous reviewers for their valuable comments, which help to improve the quality of our paper greatly.

References

1. Alur, R.: Formal verification of hybrid systems. In: EMSOFT 2011, pp. 273–278. ACM (2011)
2. Alur, R., Couroubetis, C., Henzinger, T., Ho, P.H.: Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Ravn, A.P., Rischel, H., Nerode, A. (eds.) HS 1991 and HS 1992. LNCS, vol. 736, pp. 209–229. Springer, Heidelberg (1993)
3. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. 138(1), 3–34 (1995)
4. Asarin, E., Bournez, O., Dang, T., Maler, O., Pnueli, A.: Effective synthesis of switching controllers for linear systems. Proc. of the IEEE 88(7), 1011–1025 (2000)
5. Blanchini, F.: Set invariance in control. Automatica 35(11), 1747–1767 (1999)
6. Brown, C.W.: QEPCAD B: A program for computing with semi-algebraic sets using CADs. SIGSAM Bulletin 37, 97–108 (2003)
7. Castelan, E., Hennet, J.: On invariant polyhedra of continuous-time linear systems. IEEE Trans. Autom. Control 38(11), 1680–1685 (1993)
8. Cousot, P.: Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 1–24. Springer, Heidelberg (2005)
9. Davenport, J.H., Heintz, J.: Real quantifier elimination is doubly exponential. J. Symb. Comput. 5(1-2), 29–35 (1988)
10. Dolzmann, A., Seidl, A., Sturm, T.: Redlog User Manual (November 2006), `http://redlog.dolzmann.de/downloads/`, edition 3.1, for redlog Version 3.06 (reduce 3.8)

11. Gulwani, S., Tiwari, A.: Constraint-based approach for analysis of hybrid systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 190–203. Springer, Heidelberg (2008)
12. Ho, P.H.: The algorithmic analysis of hybrid systems. Ph.D. thesis, Cornell University (1995)
13. Holmström, K., Göran, A.O., Edvall, M.M.: User's Guide for TOMLAB/PENOPT. Tomlab Optimization (November 2006), http://tomopt.com/docs/TOMLAB_PENOPT.pdf
14. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Synthesizing switching logic for safety and dwell-time requirements. In: ICCPS 2010, pp. 22–31. ACM (2010)
15. Kapur, D.: A quantifier-elimination based heuristic for automatically generating inductive assertions for programs. Journal of Systems Science and Complexity 19(3), 307–330 (2006)
16. Kapur, D.: Automatically Generating Loop Invariants Using Quantifier Elimination. Technical Report, Department of Computer Science, University of New Mexico, Albuquerque, USA (December 2003)
17. Kapur, D., Shyamasundar, R.K.: Synthesizing controllers for hybrid systems. In: Maler, O. (ed.) HART 1997. LNCS, vol. 1201, pp. 361–375. Springer, Heidelberg (1997)
18. Kapur, D., Zhan, N., Zhao, H.: Synthesizing switching controllers for hybrid systems by continuous invariant generation. CoRR abs/1304.0825 (2013), http://arxiv.org/abs/1304.0825
19. Kočvara, M., Stingl, M.: PENBMI User's Guide (Version 2.1). PENOPT GbR (March 2006), http://www.penopt.com/doc/penbmi2_1.pdf
20. Lin, W., Wu, M., Yang, Z., Zeng, Z.: Exact safety verification of hybrid systems using sums-of-squares representation. CoRR abs/1112.2328 (2011), http://arxiv.org/abs/1112.2328
21. Liu, J., Lv, J., Quan, Z., Zhan, N., Zhao, H., Zhou, C., Zou, L.: A calculus for hybrid CSP. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 1–15. Springer, Heidelberg (2010)
22. Liu, J., Zhan, N., Zhao, H.: Computing semi-algebraic invariants for polynomial dynamical systems. In: EMSOFT 2011, pp. 97–106. ACM (2011)
23. Liu, J., Zhan, N., Zhao, H.: Automatically discovering relaxed Lyapunov functions for polynomial dynamical systems. Mathematics in Computer Science 6(4), 395–408 (2012)
24. Löfberg, J.: YALMIP: A toolbox for modeling and optimization in MATLAB. In: Proc. of the CACSD Conference, Taipei, Taiwan (2004), http://users.isy.liu.se/johanl/yalmip
25. Löfberg, J.: Pre- and post-processing sum-of-squares programs in practice. IEEE Trans. Autom. Control 54(5), 1007–1011 (2009)
26. Parrilo, P.A.: Structured Semidefinite Programs and Semialgebraic Geometry Methods in Robustness and Optimization. Ph.D. thesis, California Institute of Technology, Pasadena, CA (May 2000), http://thesis.library.caltech.edu/1647/
27. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Log. and Comput. 20(1), 309–352 (2010)
28. Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 176–189. Springer, Heidelberg (2008)
29. Platzer, A.: A differential operator approach to equational differential invariants. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 28–48. Springer, Heidelberg (2012)

30. Platzer, A.: The structure of differential invariants and differential cut elimination. Logical Methods in Computer Science 8(4), 1–38 (2012)
31. Prajna, S., Jadbabaie, A.: Safety verification of hybrid systems using barrier certificates. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 477–492. Springer, Heidelberg (2004)
32. Prajna, S., Jadbabaie, A., Pappas, G.J.: A framework for worst-case and stochastic safety verification using barrier certificates. IEEE Trans. Autom. Control 52(8), 1415–1428 (2007)
33. Prajna, S., Papachristodoulou, A., Seiler, P., Parrilo, P.: SOSTOOLS and its control applications. In: Henrion, D., Garulli, A. (eds.) Positive Polynomials in Control. LNCIS, vol. 312, pp. 273–292. Springer, Heidelberg (2005)
34. Sankaranarayanan, S., Sipma, H., Manna, Z.: Non-linear loop invariant generation using Gröbner bases. In: POPL 2004 (2004)
35. Sankaranarayanan, S., Dang, T., Ivančić, F.: A policy iteration technique for time elapse over template polyhedra. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 654–657. Springer, Heidelberg (2008)
36. Sankaranarayanan, S., Dang, T., Ivančić, F.: Symbolic model checking of hybrid systems using template polyhedra. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 188–202. Springer, Heidelberg (2008)
37. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 25–41. Springer, Heidelberg (2005)
38. Sassi, M.A.B., Girard, A.: Computation of polytopic invariants for polynomial dynamical systems using linear programming. Automatica 48(12), 3114–3121 (2012)
39. Sturm, J.F.: Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. Optimization Methods and Software 11-12, 625–653 (1999)
40. Sturm, T., Tiwari, A.: Verification and synthesis using real quantifier elimination. In: ISSAC 2011, pp. 329–336. ACM (2011)
41. Taly, A., Gulwani, S., Tiwari, A.: Synthesizing switching logic using constraint solving. International Journal on Software Tools for Technology Transfer 13, 519–535 (2011)
42. Taly, A., Tiwari, A.: Deductive verification of continuous dynamical systems. In: FSTTCS 2009. LIPIcs, vol. 4, pp. 383–394 (2009)
43. Taly, A., Tiwari, A.: Switching logic synthesis for reachability. In: EMSOFT 2010, pp. 19–28. ACM (2010)
44. Tarski, A.: A Decision Method for Elementary Algebra and Geometry. University of California Press, Berkeley (1951)
45. Tomlin, C.J., Lygeros, J., Sastry, S.S.: A game theoretic approach to controller design for hybrid systems. Proc. of the IEEE 88(7), 949–970 (2000)
46. VanAntwerp, J.G., Braatz, R.D.: A tutorial on linear and bilinear matrix inequalities. Journal of Process Control 10(4), 363–385 (2000)
47. Vandenberghe, L., Boyd, S.: Semidefinite programming. SIAM Review 38(1), 49–95 (1996)
48. Yang, Z., Wu, M., Lin, W.: Exact safety verification of hybrid systems based on bilinear SOS representation. CoRR abs/1201.4219 (2012), `http://arxiv.org/abs/1201.4219`
49. Zhao, H., Zhan, N., Kapur, D., Larsen, K.G.: A "hybrid" approach for synthesizing optimal controllers of hybrid systems: A case study of the oil pump industrial example. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 471–485. Springer, Heidelberg (2012)

Graph-Based Object-Oriented Hoare Logic*

Liang Zhao[1], Shuling Wang[2,**], and Zhiming Liu[3]

[1] Institute of Computing Theory and Technology, Xidian University, Xi'an, China
[2] State Key Lab. of Comput. Sci., Institute of Software,
Chinese Academy of Sciences, Beijing, China
[3] United Nations University - International Institute for Software Technology,
Macao SAR, China
wangsl@ios.ac.cn

Abstract. We are happy to contribute to this volume of essays in honor of He Jifeng on the occasion of his 70th birthday. This work combines and extends two recent pieces of work that He Jifeng has made significant contributions: the rCOS Relational Semantics of Object-Oriented Programs [4] and the Trace Model for Pointers and Objects [7]. It presents a graph-based Hoare Logic that deals with most general constructs of object-oriented (OO) programs such as assignment, object creation, local variable declaration and (possibly recursive) method invocation. The logic is built on a graph-based operational semantics of OO programs so that assertions are formalized as properties on graphs of execution states. We believe the logic is simple because 1) the use of graphs provides an intuitive visualization of states and executions of OO programs and thus it is helpful in thinking of and formulating clear specifications, 2) the logic follows almost the whole traditional Hoare Logic and the only exception is the backward substitution law which is not valid for OO programs, and 3) the mechanical implementation of the logic would not be much more difficult than traditional Hoare Logic. Despite the simplicity, the logic is powerful enough to reason about important OO properties such as aliasing and reachability.

Keywords: Hoare Logic, object-oriented program, state graph, aliasing.

1 Introduction

Correct design of an OO program from a specification is difficult. A main reason is that the execution states of an OO program are complex, due to the complex relation among the objects, aliasing, dynamic binding and polymorphism. This makes it hard to understand, to formulate and to reason about properties on behaviors of the program. Complexity is in general the cause of breakdowns of

* The research is supported by the NSFC Grant Nos. 61133001, 61103013, 91118007 and 61100061, Projects GAVES and PEARL funded by Macao Science and Technology Development Fund, and National Program on Key Basic Research Project (973 Program) Grant No.2010CB328102.

** Corresponding author.

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 374–393, 2013.

a system and OO programs are typically prone to errors of a null pointer (or reference), an inaccessible object and aliases [7].

A formal semantic model must first contribute to *conceptual clarification* for better understanding so as to master the complexity better, and then help the thinking, formulating and reasoning about assertions of programs. To support the development of techniques and tools for analyzing and reasoning about programs, a logic is needed which should be defined based on the semantics. Obviously, a simple semantic model is essential for the definition of a logic that is easy to use for writing specifications and doing formal reasoning, and for implementing mechanical assistance.

In our earlier work [9], a graph-based operational semantics is defined and implemented for an OO programming language that is originally defined with a denotational semantics and a refinement calculus [4,18] for the rCOS method of component-based model-driven design [2,12]. In this semantics, objects of a class and execution states of a program are defined as directed and labeled graphs. A node represents an object or a simple datum. However, in the former case, the node is not labeled by an explicit reference value, but by the name of its runtime type that is a name of a class of the program. An edge is labeled by the name of a field of the source object referring to the target object. The advantage of the semantics lies in its naturalness in characterizing OO features, including the stack, heap, garbage, polymorphism and aliasing, and its intuitiveness for thinking and formulating properties of the execution of a program. Another good nature of the semantics is that it is *location independent*.

In this paper, we use the graph-based semantics to define a modest Hoare Logic for OO programs. There are mainly three OO features that make it difficult for Hoare Logic to be directly applied for specifying and reasoning about OO programs.

1. Side effects in assignment due to reference aliasing cause the invalidity of the (syntactic) backward substitution law

$$\{p[e/x]\}\, x := e\, \{p\}.$$

   For example, $\{(y.a = 4)[3/x.a]\}\, x.a := 3\, \{y.a = 4\}$ does not hold if x and y are *aliasing*, i.e. referring to the same object. There is a need of a rule for object creation which has side effects due to aliasing, too.
2. Dynamic method binding and recursions of methods make the specification of method invocations delicate.
3. Rules are needed for reasoning about dynamic typing.

In classical Hoare Logic [5], a *specification* or Hoare triple $\{p\}\, c\, \{q\}$ is defined in the way that p is a weakest precondition of the command c with respect to the postcondition q, e.g. the backward substitution law for assignment. However, the existence of aliasing makes it not so natural to propose a specification for OO commands in this way. Especially, it is very difficult to calculate a

precondition of an object creation given a postcondition that refers to the newly created object. This motivates us to take a *pre-to-post* approach, i.e. to calculate a postcondition from a precondition. Considering the example above, a correct specification should be

$$\{y.a = 4 \wedge x.a = V\}\, x.a := 3\, \{(y.a = 4)[V/x.a] \wedge x.a = 3\},$$

where we use a *logic variable* V to record the initial value of $x.a$. To deal with the problem of aliasing, we introduce a special substitution $[V/x.a]$ in the postcondition which intuitively means to substitute every term $e.a$ by V where e is an alias of x. Syntactically, $(y.a = 4)[V/x.a]$ is defined as $(V \lhd y = x \rhd y.a) = 4$, which involves a *conditional term* $V \lhd y = x \rhd y.a$. The meaning of the term is clear: it behaves as V if y is an alias of x, or as $y.a$ otherwise. If needed, we can further eliminate the auxiliary logic variable V and arrive at a more intuitive specification $\{y.a = 4\}\, x.a := 3\, \{(y.a = 3 \lhd y = x \rhd y.a = 4) \wedge x.a = 3\}$. Notice that $y.a = 3$ is implied from $y = x$ and $x.a = 3$. This is actually due to a property of aliasing: if x and y are aliasing and a value 3 is reachable from x through a navigation path a, the value 3 is reachable from y through the same navigation path a. In general, aliasing terms are identical concerning *reachability*.

In our pre-to-post approach, the specification of an object creation is straightforward. Consider a precondition p and a command $C.\mathsf{new}(x.a)$ which creates an object of a class C and makes $x.a$ refer to the object. The specification can be of the form

$$\{p \wedge x.a = V\}\, C.\mathsf{new}(x.a)\, \{p[V/x.a] \wedge \exists U \cdot (x.a = U \wedge U : C \wedge q)\}.$$

Like in the specification of an assignment, we make use of the special substitution $[V/x.a]$ so that $p[V/x.a]$ holds in the postcondition for any precondition p. Besides, we use a fresh logic variable U to refer to the newly created object, thus U is reachable from x through the navigation path a, i.e. $x.a = U$, and the *runtime type* of U is C, i.e. $U : C$. The rest part q of the postcondition says attributes of U have been initialized to their default values and U can only be accessed through the navigation path a from x in this state.

The specification of a method invocation $e.m(x; y)$ is more delicate than that of an assignment or object creation. To realize the OO mechanism of dynamic method binding, we choose the method m according to the runtime type C of e, i.e. $e : C$, instead of the type of e declared. To deal with mutually recursive methods, we take the general approach that is to assume a set of specifications of method invocations and to prove the specification of bodies of these methods based on these assumptions, e.g. in [6,1,14]. However, the assumptions made in these work often rely on actual parameters of the method invocations, which makes the proof complicated as multiple assumptions with different parameters are needed for the invocation of one method. For simplicity and also efficiency, we introduce an auxiliary command $C :: m()$ which means the general execution of a method $C :: m$, i.e. m of class C. The auxiliary command enables the assumption of *method invariants* of the form $\{p\}\, C :: m()\, \{q\}$. Such an invariant is general and capable of deriving the specification of an invocation of $C :: m$

with any actual parameters. On the other hand, the invariant itself is free of actual parameters, so only one invariant is needed for each method.

As for the problem of typing, there are two solutions. The first is to define a type system along with the logic, and the second is, similar to Lamport's TLA [11], to state correct typing as assertion and provide the rules for type checking too. To keep the simplicity of the presentation, we leave the problem of typing out of this paper, but the type system defined with the graph-based operational semantics in [10] shows that either solution could work with the logic. Another restriction in this logic is that we do not deal with attribute shadowing.

The rest of the paper is organized as follows. Section 2 briefs our notations of graphs for OO programs. Sections 3 and 4 then present the underlying assertion language and the proof system, respectively. The soundness of the logic is discussed in Section 5. Finally, conclusions are drawn with discussions on related and future work.

2 Graph Representation of OO Programs

This section summarizes the graph notations of class structures and execution states of OO programs. Details can be found in our previous work [9,10].

2.1 An OO Language

We adopt the formal language of the rCOS method [4] as the basis of our discussion. It is a general OO language with essential OO features such as object creation, inheritance, dynamic method binding, and so on.

The language is equipped with a set of primitive data types, such as *Int* and *Bool*, and a set of built-in operations $f, \cdots$ on these types. Besides, let $C, D, \cdots$ range over classes $\mathcal{C}$; $S, T, \cdots$ range over types $\mathcal{T}$, including classes and data types; $a, x, y, \cdots$ range over attributes and variables $\mathcal{A}$; $m, \cdots$ range over methods $\mathcal{M}$; and $l, \cdots$ range over literals $\mathcal{L}$, including the null reference *null*. The syntax of the language is given in Fig. 1, where text occurring in square brackets is optional and overlined text $\overline{u}$ denotes a sequence $u_1, u_2, \ldots, u_k (k \geq 0)$.

A program *prog* is a sequence of class declarations *cdecls* followed by a main method *Main* which defines the execution of the whole program. A class C is declared optionally as a *direct subclass* of another class D, thus there is no multiple inheritance. An attribute declaration *adecl* consists of its type, name and default initial value, which is a literal. An attribute declared in a class cannot be re-declared in its subclasses, i.e., we do not consider attribute shadowing. A method declaration *mdecl* consists of the method name m, its value parameters $\overline{S}\ \overline{x}$, result parameters $\overline{T}\ \overline{y}$, and body command c. Notice that a method has result parameters instead of returning values directly. This is to make sure that expressions have no side effects. As a key feature of OO, a method is allowed to be overridden in a subclass, but its signature $m(\overline{S}; \overline{T})$ must be preserved.

program	$prog$	$::= cdecls \bullet Main$
class declarations	$cdecls$	$::= cdecl \mid cdecl; cdecls$
class declaration	$cdecl$	$::= \mathsf{class}\ C\ [\mathsf{extends}\ D]\ \{\overline{adecl}; \overline{mdecl}\}$
attribute definition	$adecl$	$::= T\ a = l$
method definition	$mdecl$	$::= m(\overline{S}\ \overline{x}; \overline{T}\ \overline{y})\{c\}$
command	c	$::= \mathsf{skip} \mid le := e \mid C.\mathsf{new}(le) \mid \mathsf{var}\ T\ x\ [= e] \mid \mathsf{end}\ x$ $\mid e.m(\overline{e}; \overline{le}) \mid C :: m() \mid c; c \mid c \lhd b \rhd c \mid b * c$
expression	e	$::= le \mid \mathsf{self} \mid l \mid f(\overline{e})$
l-expression	le	$::= x \mid e.a$
boolean expression	b	$::= e \mid e = e \mid \neg b \mid b \wedge b$
main method	$Main$	$::= (\overline{ext}; c)$
external variable	ext	$::= T\ x = l$

Fig. 1. Syntax of rCOS language

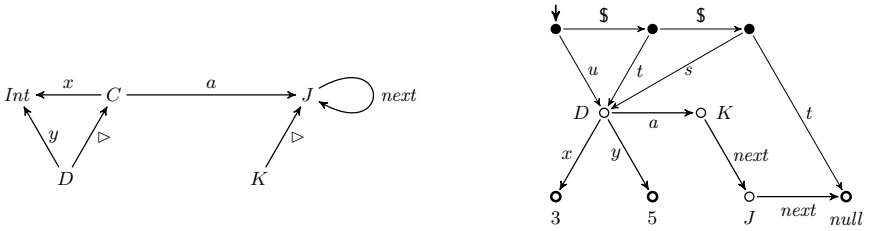
A command can be simply skip that does not do anything; $le := e$ that assigns e to le; $C.\mathsf{new}(le)$ that creates an object of class C and attaches it to le; $\mathsf{var}\ T\ x = e$ that declares a local variable x of type T with initial value e, where e is by default the zero value $zero(T)$ of T; $\mathsf{end}\ x$ that ends the scope of x; or $e.m(\overline{e}; \overline{le})$ that invokes the method m of the object e refers to, with actual value parameters $\overline{e}$ and actual result parameters $\overline{le}$. Commands for sequential composition $c_1; c_2$, conditional choice $c_1 \lhd b \rhd c_2$ and loop $b * c$ are also allowed. In addition, we introduce an auxiliary command $C :: m()$ to represent the general execution of the method $C :: m$, i.e. m defined in class C. Such a command will be used for the specification of *method invariants*.

Expressions include *assignable expressions* le, or simply l-expressions; the special self variable that represents the currently active object; literals l; and expressions $f(\overline{e})$ constructed with operations f of data types. Notice that expressions of the language have no side effects.

2.2 Class Graph and State Graph

The class declarations of a program can be represented as a directed and labeled graph, called a *class graph* [9]. In a class graph, a node represents a type T, which is either a class or a data type. There are two kinds of edges. An *attribute edge* $C \xrightarrow{a} T$, which is labeled by an attribute name a, represents that C has an attribute a of type T, while an *inheritance edge* $C \xrightarrow{\rhd} D$, which is labeled by a designated symbol $\rhd$, represents that C is a direct subclass of D. Notice that the source of an edge and the target of an inheritance edge must be nodes of classes. An example of class graph is shown in Fig. 2(1).

Given a class graph, we use $C \rhd D$ to denote C is a direct subclass of D, and $\preccurlyeq$ the subclass relation that is the reflexive and transitive closure of $\rhd$. We also use $Attr(C)$ to denote the set of attributes of C, including those inherited from C's superclasses. For an attribute $a \in Attr(C)$, we use $init(C, a)$ to denote its initial value. Besides, we introduce two partial functions $mtype(C :: m)$



(1) A class graph (2) A state graph

Fig. 2. Class graph and state graph

and $mbody(C :: m)$ for looking up the signature and the body of a method m of a class C, respectively.

$$mtype(C :: m) \mathrel{\widehat{=}} \begin{cases} (\overline{S}; \overline{T}) & \text{if } m(\overline{S}\ \overline{x}; \overline{T}\ \overline{y})\{c\} \text{ is defined in } C \\ mtype(D :: m) & \text{otherwise, if } C \rhd D \end{cases}$$

$$mbody(C :: m) \mathrel{\widehat{=}} \begin{cases} (\overline{x}; \overline{y}; c) & \text{if } m(\overline{S}\ \overline{x}; \overline{T}\ \overline{y})\{c\} \text{ is defined in } C \\ mbody(D :: m) & \text{otherwise, if } C \rhd D \end{cases}$$

With these functions, a class graph is used for type checking [10]. In addition, the class graph of an OO program is regarded as an abstract type whose instances are graphs representing executions states of the program, called *state graphs*.

Let $\mathcal{N}$ be an infinite set of node names and consider $\mathcal{A}^+ \mathrel{\widehat{=}} \mathcal{A} \cup \{\mathsf{self}, \$\}$ as the set of edge labels.

Definition 1 (State graph). *A* state graph *is a rooted, directed and labeled graph* $G = \langle N, E, \rho_{\mathsf{t}}, \rho_{\mathsf{v}}, r \rangle$*, where*

- $N \subseteq \mathcal{N}$ *is the set of nodes, denoted by* $G.node$*,*
- $E \subseteq N \times \mathcal{A}^+ \times N$ *is the set of edges, denoted by* $G.edge$*,*
- $\rho_{\mathsf{t}} : N \rightharpoonup \mathcal{C}$ *is a partial function from nodes to types, denoted by* $G.type$*,*
- $\rho_{\mathsf{v}} : N \rightharpoonup \mathcal{L}$ *is a partial function from nodes to values, denoted by* $G.value$*,*
- $r \in N$ *is the root of the graph, i.e. without incoming edges, denoted by* $G.root$*,*
- *starting from* r*, the* $\$$*-edges, if there are any, form a path such that except for* r *each node on the path has only one incoming edge.*

A state graph is a snapshot of the state at one time of the program execution, consisting of the existing objects, their attributes, as well as variables of different scopes that refer to these objects. Specifically, a state graph G has three kinds of nodes: *object nodes*, *value nodes* and *scope nodes*, representing objects, values and scopes, respectively. Object nodes are the domain of $G.type$ so that each object node is labeled by its class with outgoing edges representing its attributes. Value nodes are the domain of $G.value$ so that each value node is labeled by a value. We assume a value node is in the state graph when needed, as otherwise it can always be added. A scope node has outgoing edges representing variables declared in the scope. In addition, scope nodes are associated with a $\$$-labeled path and they constitute the *stack* of the state graph. The first (scope) node of

the stack, i.e. the source of the $-labeled path, represents the scope of the current execution. It is the *root* node of the graph through which variables and objects of the state can be accessed. An example state graph is shown in Fig. 2(2).

To represent a sound state, a state graph G should satisfy a few conditions of *well-formedness* [9], e.g. outgoing edges of each node have distinct labels. In addition, a state graph should be *correctly typed* with respect to the class graph of a program. Intuitively, the class of each object node is defined in the class graph and each attribute is correctly typed according to the class graph. For example, the state graph in Fig. 2(2) is correctly typed with respect to the class graph in Fig. 2(1).

Trace and evaluation. We use the term *trace*, or navigation path, to denote a sequence of edge labels. In a state graph, every path $G.root \xrightarrow{x_1} n_1 \xrightarrow{x_2} \cdots \xrightarrow{x_k} n_k$ from the root is uniquely determined by its trace $x_1.x_2.\cdots.x_k$. We thus allow the interchange between a root-originating path and its trace. Besides, we do not distinguish state graphs different only in the choice of their node names, and this is formalized by the notion of graph isomorphism [9]. Notice that isomorphic state graphs have the same set of traces.

Given a state graph G that represents a state, the evaluation of an expression e returns its value $eval(e)$ and runtime type $rtype(e)$ in the state [9]. For most expressions e, the evaluation is simply the calculation of their traces $trace(e)$. If the trace of e targets at an object node o, $eval(e) \mathrel{\widehat{=}} o$ and $rtype(e) \mathrel{\widehat{=}} G.type(o)$. Otherwise, $eval(e) \mathrel{\widehat{=}} G.value(v)$ which is a literal and $rtype(e) \mathrel{\widehat{=}} \mathbf{T}(G.value(v))$. Here, $\mathbf{T}(l)$ denotes the type of a literal l. Notice that the trace of an expression e may not exist. In this case, the evaluation fails and we denote both $eval(e)$ and $rtype(e)$ as $\perp$. To sum up, every expression evaluates to an element in $\mathcal{V} \mathrel{\widehat{=}} \mathcal{N} \cup \mathcal{L} \cup \{\perp\}$. Thus we call $\mathcal{V}$ the *value space*.

2.3 Graph Operation

We defined an operational semantics of the OO language in terms of transitions $con \to con$ between *configurations* [9]. Here, a configuration con is either $\langle c, G\rangle$ representing a command c to be executed and a state G, or G representing the state that the execution terminates at. The semantics is simple in the sense that it is defined by a few basic operations on state graphs.

Swing. The most frequent operation on a state graph is an edge swing. Specifically, for an edge $d = v_1 \xrightarrow{a} v_2$ and a node v of G, $swing(G, d, v)$ is the graph obtained from G by making d target at v (instead of v_2). The swing $swing(G, \alpha, v)$ of a trace α is the swing of its last edge, see Fig. 3. The swing operation is used to define the semantics of an assignment: $\langle le := e, G\rangle \to swing(G, trace(le), eval(e))$.

New. Given a state graph G, a class C and a trace α in G, the operation $new(G, C, \alpha)$ creates an object node of class C with attributes initialized by default values and then swings α to the new object node, see Fig. 4. This operation is used to define the semantics of object creation: $\langle C.\mathsf{new}(le), G\rangle \to new(G, C, trace(le))$.

Push and pop. Let G be a state graph, x a variable and v a node of G. The operation $push(G, x, v)$ adds a new scope node r, with an outgoing edge labeled

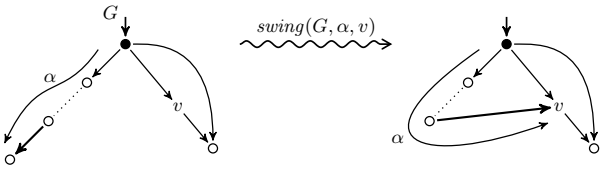


Fig. 3. Trace swing

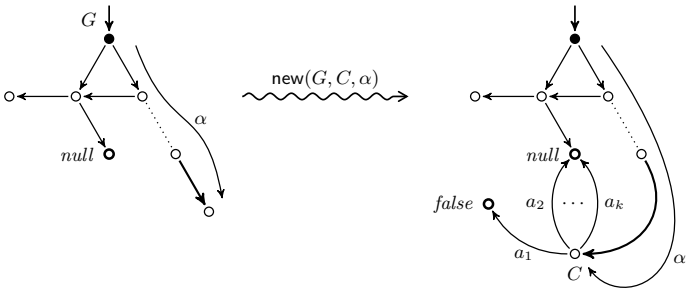


Fig. 4. Object creation

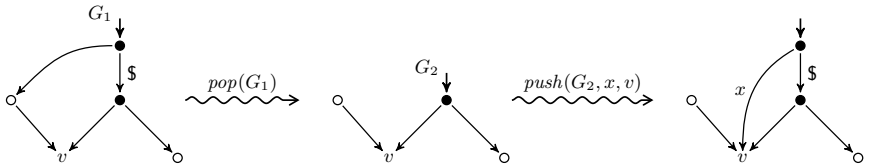


Fig. 5. Stack push and pop

by x and targeting at v, to the top of the stack so that r becomes the root of the result graph. In contrast, the operation $pop(G)$ removes the root node together with its outgoing edges from the graph, while the next scope node becomes the root. They are shown in Fig. 5. The push operation is used to define the declaration of a local variable: $\langle \mathsf{var}\ T\ x = e, G\rangle \to push(G, x, eval(e))$, as well as the switch of the execution into the method body at the beginning of a method invocation. Correspondingly, the pop operation is used to define the un-declaration of a local variable: $\langle \mathsf{end}\ x, G\rangle \to pop(G)$, as well as the switch of the execution out of the method body at the end of a method invocation.

3 Assertion Language

The advantage of our graph notations lies in both the intuitive understanding and the theoretical maturity of graphs. They are thus helpful to formulate clear and precise assertions on the execution of OO programs. In this section, we propose an assertion language as the basis of our Hoare Logic. It is a first-order language with equality characterizing the aliasing property.

$$
\begin{array}{lll}
\text{assertion} & p ::= & P(\overline{t}) \mid t = t \mid t\uparrow \mid t : C \\
 & & \mid \mathit{true} \mid \mathit{false} \mid \neg p \mid p \wedge p \mid \exists U \cdot p \\
\text{term} & t ::= & x \mid t.a \mid \mathsf{self} \mid l \mid f(\overline{t}) \\
 & & \mid U \mid t \triangleleft t = t \triangleright t
\end{array}
$$

Fig. 6. Syntax of the assertion language

Let $\mathcal{O}$ be the vocabulary of *logic variables* $U, V, \cdots$ and let $P, \ldots$ range over predicates. The syntax of the assertion language is given in Fig. 6. Assertions include $P(\overline{t})$ that applies a k-ary predicate P on a sequence of k terms $\overline{t}$; $t_1 = t_2$ that says terms t_1 and t_2 are *aliasing*; $t\uparrow$ that claims t successfully evaluates to a value not $\perp$; and $t : C$ that asserts the runtime type of t is class type C. General constructs of a first-order language are also allowed, such as negation $\neg p$, conjunction $p_1 \wedge p_2$ and (existential) quantification $\exists U \cdot p$. We regard $p_1 \Rightarrow p_2$, $p_1 \vee p_2$, $p_1 \Leftrightarrow p_2$, $\forall U \cdot p$ as shorthands for $\neg(p_1 \wedge \neg p_2)$, $\neg p_1 \Rightarrow p_2$, $(p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_1)$, $\neg \exists U \cdot \neg p$, respectively.

The syntax of terms is simply an extension of that of expressions (see Fig. 1) with logic variables and conditional terms. A logic variable U is introduced to record a constant value. This value cannot be changed by the execution of commands since U never occurs in a command. A conditional term $t_1 \triangleleft t = t' \triangleright t_2$ behaves as t_1 or t_2, depending on whether t and t' are aliasing or not. We use $lv(t)$ to denote the set of logic variables that occur in a term t, and $\mathit{flv}(p)$ the set of logic variables that occur *free*, i.e. not bound by quantifiers, in an assertion p.

3.1 Satisfaction of Assertion

As for the semantics of the assertion language, we characterize whether an assertion p is satisfied by a state graph G. Since assertions contain logic variables, we extend the notion of state graph with logic variables correspondingly.

Definition 2 (Extended state graph). *An* extended state graph *is a rooted, directed and labeled graph* $G = \langle N, E, \rho_{\mathsf{c}}, \rho_{\mathsf{v}}, r, v \rangle$*, where*

- N*,* ρ_{c}*,* ρ_{v} *and* r *are defined as in Definition 1,*
- $E \subseteq N \times (\mathcal{A}^+ \cup \mathcal{O}) \times N$ *is the set of edges, denoted by* $G.edge$*,*
- $v \in N$ *is a special node without incoming edges, denoted by* $G.lvar$*, such that each outgoing edge of* v *is labeled by a logic variable, and furthermore, each edge labeled by a logic variable is an outgoing edge of* v*.*

The extension to an original state graph G is mainly a special node $G.lvar$ with outgoing edges $G.lvar \xrightarrow{U_1} n_1, \ldots, G.lvar \xrightarrow{U_k} n_k$ recording a set of logic variables. We use $lv(G)$ to denote the set of logic variables $\{U_1, \ldots, U_k\}$ of G.

Notice that all the graph operations provided in Section 2.3, and thus the operational semantics, are applicable to extended state graphs. In fact, the execution of a command never changes the existence and values of logic variables. In the rest of the paper, a state graph always means an extended one.

Given a term t and a state graph G with $lv(t) \subseteq lv(G)$, the evaluation of t calculates the value $\mathit{eval}(t)$ and the runtime type $\mathit{rtype}(t)$ of t in G. For a logic

variable $U \in lv(G)$, there must be an edge $G.lvar \xrightarrow{U} n$ in G. If n is an object node, $eval(U) \mathrel{\widehat{=}} n$ and $rtype(U) \mathrel{\widehat{=}} G.type(n)$. If n is a value node, $eval(U) \mathrel{\widehat{=}} G.value(n)$ and $rtype(U) \mathrel{\widehat{=}} \mathbf{T}(G.value(n))$. The evaluation of a conditional term $t \equiv t_1 \lhd t' = t'' \rhd t_2$ is the same as that of t_1 or t_2, depending on whether $eval(t') = eval(t'')$ or not. Other constructs of terms evaluate in the same way as those of expressions.

To reason about the satisfaction of predicates, we consider the notion of *interpretation*. An interpretation I of the assertion language interprets every k-ary predicate P as a k-ary relation on the value space $\mathcal{V}$, i.e. $I(P) \subseteq \mathcal{V}^k$. To calculate the satisfaction of a quantified assertion $\exists U \cdot p$, we introduce an operation that adds a logic variable U into a state graph G and makes it refer to a node n of G:

$$addv(G, U, n) \mathrel{\widehat{=}} G' \qquad \text{provided } U \notin lv(G),$$

where G' is the same as G except that $G'.edge = G.edge \cup \{G.lvar \xrightarrow{U} n\}$. Intuitively, $\exists U \cdot p$ is satisfied by G if there is an object node or value node n of G such that p is satisfied by $addv(G, U, n)$. The satisfaction of other assertions can be defined straightforwardly. For example, $t_1 = t_2$ is satisfied if t_1 and t_2 evaluate to the same value, $t\uparrow$ is satisfied if t evaluates to a value other than $\perp$, while $t : C$ is satisfied if the runtime type of t is C.

Definition 3 (Satisfaction of assertion). *For an assertion p, an interpretation I and a state graph G with $flv(p) \subseteq lv(G)$, we use $G \models_I p$ to denote that p is* satisfied *by G under I. It is defined inductively on the structure of p.*

- *For $p \equiv P(t_1, \ldots, t_k)$, $G \models_I p$ if $(eval(t_1), \ldots, eval(t_k)) \in I(P)$.*
- *For $p \equiv t_1 = t_2$, $G \models_I p$ if $eval(t_1) = eval(t_2)$.*
- *For $p \equiv t\uparrow$, $G \models_I p$ if $eval(t) \neq \perp$.*
- *For $p \equiv t : C$, $G \models_I p$ if $rtype(t) = C$.*
- *For $p \equiv true$, $G \models_I p$ always holds; for $p \equiv false$, $G \models_I p$ never holds.*
- *For $p \equiv \neg p_1$, $G \models_I p$ if $G \models_I p_1$ does not hold.*
- *For $p \equiv p_1 \wedge p_2$, $G \models_I p$ if $G \models_I p_1$ and $G \models_I p_2$,*
- *For $p \equiv \exists U \cdot p_1$, assume $U \notin lv(G)$ as U can be renamed by alpha-conversion. $G \models_I p$ if $addv(G, U, n) \models_I p_1$ for some object node or value node n of G.*

We say p is true *under I, denoted as $\models_I p$, if $G \models_I p$ for any state graph G with $flv(p) \subseteq lv(G)$. In addition, we say p is* valid*, denoted as $\models p$, if $\models_I p$ under any interpretation I.*

Notice that we use $\equiv$ to denote the equivalence in syntax. For example, $p \equiv t : C$ means p and $t : C$ represent the same syntactic assertion, i.e., p is exactly $t : C$.

It is straightforward to verify that the satisfaction of an assertion does not rely on the naming of nodes of the underlying state graph, i.e., $G \models_I p$ if and only if $G' \models_I p$ for isomorphic state graphs G and G'. This indicates that our OO assertion model, and further the proof system, is *location independent*.

The semantics of assertions provided in the above definition is consistent with the semantics of first-order logic. As a result, every valid formula of first-order

logic, e.g. $\neg\neg p \Leftrightarrow p$, is a valid assertion. In addition, it is straightforward to prove the validity of the following OO assertions, where $t \neq t'$ is a shorthand for $\neg t = t'$.

1. $t = t' \Rightarrow t_1 = t'_1$, provided t'_1 is obtained from t_1 by replacing one or more occurrence of t by t'. This assertion says that aliasing terms share the same properties.
2. $t.a \uparrow \Rightarrow t \uparrow \wedge\, t \neq \mathit{null}$. This assertion says that the evaluation of $t.a$ successes only if t evaluates to a non-*null* object.
3. $t : C \Rightarrow t \uparrow \wedge\, t \neq \mathit{null}$. This assertion says that only objects can have runtime class types.
4. $t \uparrow \Leftrightarrow \exists U \cdot U = t$ provided U is fresh. This assertion reflects the intuition that a logic variable is used to record a value.

4 Proof System

The assertion language enables us to define program *specifications*. A specification takes the form $\{p\}\, c\, \{q\}$, where p, q are assertions and c is a command. We call p and q the *precondition* and *postcondition* of the specification, respectively. Intuitively, such a specification means that if p holds before c executes, and when the execution of c terminates, then q holds after the execution. We will formally define the semantics of specifications in the next section. For a specification $\{p\}\, c\, \{q\}$, we always assume $\mathit{flv}(q) \subseteq \mathit{flv}(p)$. In fact, a specification that generates new free logic variables in the postcondition is not necessary. For example, by $\{x = y\}\, \mathsf{skip}\, \{x = V \wedge V = y\}$, we actually mean $\{x = y\}\, \mathsf{skip}\, \{\exists V \cdot x = V \wedge V = y\}$.

In this section, we present the Hoare proof system that consists of a set of *logic rules* for specifications of all constructs of the OO language presented in Section 2.1. Each logic rule defines a one-step proof (or derivation) of a *conclusion* from zero or more *hypotheses*. The conclusion takes the form of a specification, while each hypothesis can be either an assertion, a specification, or a *specification sequent* $\Phi \vdash \Psi$, where Φ and Ψ are sets of specifications. The specification sequent means Ψ can be proved (or derived) from Φ by applying the logic rules.

For natural specification of commands such as assignment and object creation, we define the proof system in a *pre-to-post* way. That is, each logic rule calculates the postcondition of a specification from an arbitrary precondition.

4.1 Assignment

For specification of an assignment $le := e$, we introduce two logic variables V_0 and V to record the values of le and e before the assignment, respectively. The l-expression le can be either a variable x or a navigation expression $e_0.a$.

In the former case $le \equiv x$, the specification is straightforward and $x = V$ holds in the postcondition.

$$\{p \wedge x = V_0 \wedge e = V\}\ x := e\ \{p[V_0/x] \wedge x = V\} \tag{1}$$

If the precondition p does not contain x, p also holds in the postcondition because the assignment only modifies the value of x. Otherwise, we can replace each x in p by its original value V_0, so that $p[V_0/x]$ holds in the postcondition.

In the latter case $le \equiv e_0.a$, we use an extra logic variable U to record the value of e_0, so that $U.a = V$ holds in the postcondition.

$$\{p \wedge e_0 = U \wedge U.a = V_0 \wedge e = V\}\ e_0.a := e\ \{p[V_0/U.a] \wedge U.a = V\} \tag{2}$$

For the rest of the postcondition, we introduce a special substitution $[V/U.a]$. Intuitively, $p[V/U.a]$ is obtained from p by replacing every (sub-)term of the form $t'.a$, where t' is an alias of U, by the logic variable V. As a result, the satisfaction of $p[V/U.a]$ is not compromised by the assignment. This substitution is formally defined according to the structure of assertions p, as well as terms t.

$$p[V/U.a] \mathrel{\widehat{=}} \begin{cases} P(t_1[V/U.a], \ldots, t_k[V/U.a]) & \text{if } p \equiv P(t_1, \ldots, t_k) \\ t_1[V/U.a] = t_2[V/U.a] & \text{if } p \equiv t_1 = t_2 \\ t[V/U.a] \uparrow & \text{if } p \equiv t \uparrow \\ t[V/U.a] : C & \text{if } p \equiv t : C \\ p & \text{if } p \equiv \mathit{true} \text{ or } \mathit{false} \\ \neg p_1[V/U.a] & \text{if } p \equiv \neg p_1 \\ p_1[V/U.a] \wedge p_2[V/U.a] & \text{if } p \equiv p_1 \wedge p_2 \\ \exists V' \cdot p_1[V/U.a] & \text{if } p \equiv \exists V' \cdot p_1 \text{ where } V' \text{ is not } U \text{ or } V \end{cases}$$

$$t[V/U.a] \mathrel{\widehat{=}} \begin{cases} t & \text{if } t \equiv x, \mathsf{self}, l \text{ or } V \\ t_1[V/U.a].a_1 & \text{if } t \equiv t_1.a_1,\ a_1 \not\equiv a \\ V \triangleleft t_1[V/U.a] = U \triangleright t_1[V/U.a].a & \text{if } t \equiv t_1.a \\ f(t_1[V/U.a], \ldots, t_k[V/U.a]) & \text{if } t \equiv f(t_1, \ldots, t_k) \\ t_1[V/U.a] \triangleleft t'[V/U.a] = t''[V/U.a] \triangleright t_2[V/U.a] & \text{if } t \equiv t_1 \triangleleft t' = t'' \triangleright t_2 \end{cases}$$

4.2 Object Creation

For specification of an object creation $C.\mathsf{new}(le)$, we use a logic variables V_0 to record the original value of le. Like in the specification of assignment, we need to consider two cases of le: a variable x, or a navigation expression $e_0.a$.

For $le \equiv x$, the logic rule is given as follows.

$$\begin{array}{l} \text{provided } V \text{ is fresh} \\ \{p \wedge x = V_0\}\ C.\mathsf{new}(x) \\ \{p[V_0/x] \wedge \exists V \cdot x = V \wedge V : C \wedge V = C_{init} \wedge V \neq p[V_0/x]\} \end{array} \tag{3}$$

Similar to Rule (1), $p[V_0/x]$ holds in the postcondition given any precondition p. In addition, we introduce a fresh logic variable V, which is existentially quantified, in the postcondition to record the reference to the new object of class C, thus $x = V$ and $V : C$ hold. For the rest of the postcondition, $V = C_{init}$ says that the attributes of the new object are initialized, while $V \neq p[V_0/x]$ indicates that the new object can only be accessed from x but not $p[V_0/x]$.

Formally, $V = C_{init}$ is a shorthand for $V.a_1 = init(C, a_1) \wedge \ldots \wedge V.a_k = init(C, a_k)$, provided $Attr(C) = \{a_1, \ldots, a_k\}$. $V \neq p$ is a shorthand for $V \neq t_1 \wedge \ldots \wedge V \neq t_k$, where $t_1, \ldots, t_k$ are the *free maximum* terms occurring in p. A term is free if it does not contain a quantified logic variable, while a term is maximum if it does not occur as a sub-term of another term.

For $le \equiv e_0.a$, we use a logic variable U to record the original value of e_0.

$$\begin{array}{l}\text{provided } V \text{ is fresh}\\ \{p \wedge e_0 = U \wedge U.a = V_0\}\, C.\mathsf{new}(e_0.a)\\ \{p[V_0/U.a] \wedge \exists V \cdot U.a = V \wedge V : C \wedge V = C_{init} \wedge V \neq p[V_0/U.a]\}\end{array} \tag{4}$$

This rule is similar to Rule (3), while $p[V_0/U.a]$ holds in the postcondition.

4.3 Local Variable Declaration

We only consider the specification of $\mathsf{var}\ T\ x = e; c; \mathsf{end}\ x$ where x is initialized by e, because $\mathsf{var}\ T\ x; c; \mathsf{end}\ x$ is a shorthand for $\mathsf{var}\ T\ x = zero(T); c; \mathsf{end}\ x$. We use a logic variable V to record the original value of e, so that the specification of $\mathsf{var}\ T\ x = e; c; \mathsf{end}\ x$ depends on a specification of c with $x = V$ in the precondition.

If x does not occur in a precondition p or the expression e, we can simply use p as a precondition of c which will lead to a postcondition q. To obtain the overall postcondition that holds after the execution of $\mathsf{end}\ x$, we hide all occurrences of x in q by existential quantification.

$$\frac{\text{provided } U \text{ is fresh}\quad \{p \wedge x = V\}\, c\, \{q\}}{\{p \wedge e = V\}\, \mathsf{var}\ T\ x = e; c; \mathsf{end}\ x\, \{\exists U \cdot q[U/x]\}}$$

If x occurs in p or e, we need to record the value of x by a logic variable W so as to recover x after the execution of $\mathsf{var}\ T\ x = e; c; \mathsf{end}\ x$. Of course, p cannot be used as a precondition of c until we replace each occurrence of x by W.

$$\frac{\text{provided } U \text{ is fresh}\quad \{p[W/x] \wedge x = V\}\, c\, \{q\}}{\{p \wedge e = V \wedge x = W\}\, \mathsf{var}\ T\ x = e; c; \mathsf{end}\ x\, \{(\exists U \cdot q[U/x]) \wedge x = W\}}$$

For conciseness, we unify the above two cases into a single logic rule.

$$\frac{\text{provided } U \text{ is fresh; let } p_* \text{ be } p \wedge e = V\quad \{p[W/x] \wedge x = V\}\, c\, \{q\}}{\{p_* \wedge (?p_*)x = W\}\, \mathsf{var}\ T\ x = e; c; \mathsf{end}\ x\, \{(\exists U \cdot q[U/x]) \wedge (?p_*)x = W\}} \tag{5}$$

Here, $(?p)w = t$ is a designated assertion defined as follows, in which w is either a variable or self. We will also use it in the specification of method invocations.

$$(?p)w = t \mathrel{\widehat{=}} \begin{cases} w = t & \text{if } w \text{ occurs in } p \\ true & \text{otherwise} \end{cases}$$

4.4 Method Invocation

A key feature of OO programs is the dynamic binding of method invocation. That is, a method invocation $e.m(ve; re)$ is an invocation of the method $C :: m$ where C is the runtime type of e. In our logic, the condition "C is the runtime type of e" is naturally characterized by an assertion $e : C$.

For specification of an invocation of $C :: m$, we make use of a *method invariant* $\{p\}\, C :: m()\, \{q\}$ that is a specification of the general execution of the method $C :: m()$. Specifically, the semantics of $C :: m()$ is defined the same as that of the body command of $C :: m$. Therefore, if a method $C :: m$ is non-recursive, its invariant is directly proved from the specification of its body command.

$$\frac{\begin{array}{l}\text{provided } \mathit{mbody}(C :: m) = (x; y; c)\\ \{p\}\, c\, \{q\}\end{array}}{\{p\}\, C :: m()\, \{q\}} \tag{6}$$

Once a method invariant is proved, it is used to derive specifications of invocations $e.m(ve; re)$ of the method $C :: m$ with any (well-typed) actual parameters $(ve; re)$, where $e : C$.

$$\frac{\begin{array}{l}\text{provided } \mathit{mtype}(C::m) = (S; T);\ \mathit{mbody}(C::m) = (x; y; c);\ W_4,W_5,W_6,W_7 \text{ fresh;}\\ \text{let } p_* \text{ be } p \wedge e = U \wedge ve = V \wedge re \uparrow\\ \{p[W_1, W_2, W_3/\mathsf{self}, x, y] \wedge \mathsf{self} = U \wedge x = V \wedge y = \mathit{zero}(T)\}\, C :: m()\, \{q\}\end{array}}{\begin{array}{l}\{p_* \wedge U : C \wedge \underline{re}? = V_0 \wedge (?p_*)\mathsf{self} = W_1 \wedge (?p_*)x = W_2 \wedge (?p_*)y = W_3\}\\ e.m(ve; re)\, \{(\exists W_4, W_5, W_6, W_7 \cdot q[W_4, W_5, W_6, W_7/\mathsf{self}, x, y, re[V_0/\_?]]\\ \quad \wedge re[V_0/\_?] = W_6) \wedge (?p_*)\mathsf{self} = W_1 \wedge (?p_*)x = W_2 \wedge (?p_*)y = W_3\}\end{array}} \tag{7}$$

Given a precondition p of the method invocation, we use logic variables U and V to record respectively the values of e and the value parameter ve in p, so that U and V are respectively the values of self and x in the precondition of $C :: m()$. Besides, the result parameter re must have a value to receive the result of the invocation, thus $p_* \equiv p \wedge e = U \wedge ve = V \wedge re \uparrow$ is part of the precondition of the invocation. If p_* contains self, x and y, we record their values by logic variables W_1, W_2 and W_3, respectively, and recover them after the invocation. However, p_* may not contain all of them. In the case p_* contains self and y but not x, for example, we only need to introduce the corresponding logic variables W_1 and W_3. To unify different cases, we make use of the notation $(?p_*)w = t$ defined in Section 4.3. This is similar to Rule (5) for local variable declaration. The rest of the postcondition is obtained from that of $C :: m()$ by hiding self, x and y that are local to $C :: m()$. For this, we first replace them by fresh logic variables W_4, W_5 and W_6, respectively, thus W_6 actually records the result of the invocation. Then, we hide these logic variables with existential quantification.

The rest of the rule is the return of result of the invocation W_6 to the result parameter re. There are two cases of re: a variable x or a navigation expression $e_0.a$. In the former case, we simply return W_6 to x. In the latter case, we introduce an extra logic variable V_0 to record the *parent object* e_0 of re before the invocation and return W_6 to $V_0.a$ after the invocation. This is the so-called *early binding*

of result parameters. For unification of the two cases, we introduce a designated assertion $\underline{le}? = V$ and a designated term $le[V/\_?]$ for l-expressions le and logic variables V.

$$\underline{le}? = V \mathrel{\widehat{=}} \begin{cases} e = V & \text{if } le \equiv e.a \\ true & \text{if } le \equiv x \end{cases} \qquad le[V/\_?] \mathrel{\widehat{=}} \begin{cases} V.a & \text{if } le \equiv e.a \\ x & \text{if } le \equiv x \end{cases}$$

Of course, we need to hide the value of $re[V_0/\_?]$ before returning W_6 to $re[V_0/\_?]$.

To avoid unnecessary name conflicts, we always assume that the result parameter re of a method invocation $e.m(ve; re)$ has a different name from a formal parameter x of a method declaration. Otherwise, e.g. $re \equiv x$, we can replace $e.m(ve; x)$ by an equivalent command $\mathsf{var}\ T\ z; e.m(ve; z); x := z; \mathsf{end}\ z$, where T is the type of the result parameter of m and z is a fresh name.

Recursive method invocation. Rule (6) is not strong enough to prove the invariants of recursive methods. For example, if the body c of a method $C :: m$ involves an invocation of $C :: m$ itself, the hypothesis $\{\ldots\}\, c\, \{\ldots\}$ of the rule would in turn rely on the conclusion $\{\ldots\}\, C :: m()\, \{\ldots\}$ of the rule and thus cannot be proved.

We take the general approach to dealing with recursion [6,1]. Assume a group of mutually recursive methods $C_1 :: m_1, \ldots, C_k :: m_k$, each of which may call the whole group in its body. If the specifications of the method bodies $\{p_1\}\, c_1\, \{q_1\}; \ldots; \{p_k\}\, c_k\, \{q_k\}$ can be proved under assumptions of the invariants $\{p_1\}\, C_1 :: m_1()\, \{q_1\}; \ldots; \{p_k\}\, C_k :: m_k()\, \{q_k\}$, these assumptions are established.

$$\frac{\begin{array}{l}\text{provided } mbody(C_i :: m_i) = (x_i; y_i; c_i) \text{ for } i = 1, \ldots, k \\ \{p_i\}\, C_i :: m_i()\, \{q_i\}_{\,i=1,\ldots,k} \vdash \{p_i\}\, c_i\, \{q_i\}_{\,i=1,\ldots,k}\end{array}}{\{p_j\}\, C_j :: m_j()\, \{q_j\}_{\,1 \le j \le k}} \tag{8}$$

4.5 Other Constructs

Logic rules of sequential composition, conditional choice and while loop simply follow the traditional Hoare Logic [5].

$$\frac{\{p\}\, c_1\, \{p_1\} \qquad \{p_1\}\, c_2\, \{q\}}{\{p\}\, c_1; c_2\, \{q\}} \tag{9}$$

$$\frac{\{p \wedge b\}\, c_1\, \{q\} \qquad \{p \wedge \neg b\}\, c_2\, \{q\}}{\{p\}\, c_1 \lhd b \rhd c_2\, \{q\}} \tag{10}$$

$$\frac{\{p \wedge b\}\, c\, \{p\}}{\{p\}\, b * c\, \{p \wedge \neg b\}} \tag{11}$$

4.6 Auxiliary Rules

Besides rules for deriving specifications of various OO constructs, we have *auxiliary rules* that are useful to transform the precondition and postcondition of

a specification. First, we can make the precondition of a specification "stronger" and the postcondition "weaker". And this is the so-called *consequence rule*.

$$\frac{\begin{array}{l}\text{provided } \mathit{flv}(p) \subseteq \mathit{flv}(p');\ \mathit{flv}(q') \subseteq \mathit{flv}(q)\\ p' \Rightarrow p \quad \{p\}\, c\, \{q\} \quad q \Rightarrow q'\end{array}}{\{p'\}\, c\, \{q'\}} \tag{12}$$

Another rule is about *constant* assertions. An assertion p is called constant if it does not contain any variable x, self or navigation expression $e.a$. The satisfaction of a constant assertion cannot be changed by the execution of commands.

$$\frac{\begin{array}{l}\text{provided } p \text{ is constant}\\ \{p\}\, c\, \{p\}\end{array}}{} \tag{13}$$

In addition, we can combine specifications of the same command by conjunction. We can also hide, by existential quantification, and rename logic variables.

$$\frac{\{p_1\}\, c\, \{q_1\} \qquad \{p_2\}\, c\, \{q_2\}}{\{p_1 \wedge p_2\}\, c\, \{q_1 \wedge q_2\}} \tag{14}$$

$$\frac{\{p\}\, c\, \{q\}}{\{\exists U \cdot p\}\, c\, \{\exists U \cdot q\}} \tag{15}$$

$$\frac{\begin{array}{l}\text{provided } U \text{ never occurs in the scope of } \exists V\\ \{p\}\, c\, \{q\}\end{array}}{\{p[V/U]\}\, c\, \{q[V/U]\}} \tag{16}$$

4.7 Example

We use an example to show the application of the proof system. Consider the following class declaration, with a recursive method $fact(Int\ x; Int\ y)$ to calculate the factorial of x and to return it to y.

$$\begin{array}{l}\mathsf{class}\ C\{\ldots;\\ \quad fact(Int\ x; Int\ y)\{\\ \quad\quad (\mathsf{var}\ Int\ z; \mathsf{self}.fact(x-1; z); y := z * x; \mathsf{end}\ z) \triangleleft x > 1 \triangleright y := 1\\ \quad \}\\ \}\end{array}$$

We are going to prove the method is correctly defined:

$$\{e : C \wedge z \uparrow\}\, e.fact(5; z)\, \{z = 5!\}.$$

For this, we need to prove an invariant (INV) of the method:

$$\{x = V \wedge V \geq 0 \wedge \mathsf{self} = U \wedge U : C \wedge y \uparrow\}\, C :: fact()\, \{x = V \wedge V \geq 0 \wedge y = V!\}.$$

This invariant is strong enough to derive the above conclusion using Rule (7) of method invocation, as well as auxiliary rules (12), (13), (14) and (15).

We use Rule (8) of recursion to prove (INV). That is, assuming (INV), we prove the following specification of the method body, denoted as (BOD).

$$\{x = V \wedge V \geq 0 \wedge \mathsf{self} = U \wedge U : C \wedge y \uparrow\}$$
$$c_1 \triangleleft x > 1 \triangleright y := 1 \{x = V \wedge V \geq 0 \wedge y = V!\}$$

where c_1 is $\mathsf{var}\ Int\ z; \mathsf{self}.fact(x-1; z); y := z * x; \mathsf{end}\ z$.

From (INV) and Rule (7) of method invocation, as well as auxiliary rules (12), (13), (14), (15) and (16), we have

$$\{x = V \wedge V \geq 0 \wedge \mathsf{self} = U \wedge U : C \wedge y \uparrow \wedge x > 1 \wedge z = V_0\}$$
$$\mathsf{self}.fact(x-1; z) \{V \geq 1 \wedge z = (V-1)! \wedge x = V \wedge y \uparrow\}.$$

From Rule (1) of assignment, as well as Rule (15), we have

$$\{V \geq 1 \wedge z = (V-1)! \wedge x = V \wedge y \uparrow\}$$
$$y := z * x \{x = V \wedge V \geq 1 \wedge y = V! \wedge z = (V-1)!\}.$$

By Rule (9) of sequential composition, the above two specifications lead to

$$\{x = V \wedge V \geq 0 \wedge \mathsf{self} = U \wedge U : C \wedge y \uparrow \wedge x > 1 \wedge z = V_0\}$$
$$\mathsf{self}.fact(x-1; z); y := z * x \{x = V \wedge V \geq 1 \wedge y = V! \wedge z = (V-1)!\}.$$

Then, using Rule (5) of local variable declaration, as well as auxiliary rules (12) and (15), we arrive at the following specification.

$$\{x = V \wedge V \geq 0 \wedge \mathsf{self} = U \wedge U : C \wedge y \uparrow \wedge x > 1\}\, c_1 \{x = V \wedge V \geq 0 \wedge y = V!\}$$

On the other hand, we use Rule (1) of assignment, as well as auxiliary rules (12) and (15), and arrive at the following specification.

$$\{x = V \wedge V \geq 0 \wedge \mathsf{self} = U \wedge U : C \wedge y \uparrow \wedge \neg(x > 1)\}\, y := 1 \{x = V \wedge V \geq 0 \wedge y = V!\}$$

Finally, (BOD) is proved from the above two specifications using Rule (10) of conditional choice.

5 Soundness of the Logic

We have provided a Hoare proof system for specification of OO programs, denoted as $\mathcal{H}$, which consists of a set of logic rules (1) to (16). In this section, we show that $\mathcal{H}$ is sound. For this, we need to define the semantics of specifications.

Let $\{p\}\, c\, \{q\}$ be a specification and I be an interpretation of assertions. We say $\{p\}\, c\, \{q\}$ is *true* under I, denoted as $\models_I \{p\}\, c\, \{q\}$, if for any state graphs G, G', $G \models_I p$ and $\langle c, G\rangle \rightarrow^* G'$ imply $G' \models_I q$. We call $\{p\}\, c\, \{q\}$ *valid*, denoted as $\models \{p\}\, c\, \{q\}$, if $\models_I \{p\}\, c\, \{q\}$ under any interpretation I.

We say a specification sequent $\Phi \vdash \Psi$ is *true* under an interpretation I, if Φ is true under I implies Ψ is true under I. Naturally, a set of specifications Φ is true under I means each specification of Φ is true under I.

Recall that a hypothesis of a logic rule is either an assertion, a specification or a specification sequent. We establish the soundness of logic rules of $\mathcal{H}$ by the following theorem.

Theorem 1 (Soundness of Logic Rules). *Rules (1) to (16) are sound. Here, a logic rule is* sound *means: for any interpretation I, the hypotheses of the rule are true under I implies the conclusion of the rule is true under I.*

Notice that the soundness of a logic rule with no hypothesis simply means the validity of the specification as the conclusion of the rule. The proof of this theorem can be found in our technical report [19].

As a natural deduction of Theorem 1, the proof system $\mathcal{H}$ is *sound*. That is, every specification proved by $\mathcal{H}$ is valid.

Theorem 2 (Soundness). $\vdash \{p\}\, c\, \{q\}$ *implies* $\models \{p\}\, c\, \{q\}$.

6 Conclusions

We propose a graph-based Hoare Logic for reasoning about OO programs. Specifically, the Hoare proof system consists of a set of logic rules that covers most OO constructs such as object creation, local variable declaration and recursive method invocation. We have proved the soundness of the logic that every specification proved by the system is valid.

A distinct feature of the logic is its underlying graph-based operational semantics where execution states of OO programs are visualized as directed and labeled graphs [9]. The simplicity and intuitiveness of graphs improve people's understanding of OO concepts and are thus helpful in thinking of and formulating clear assertions. On the other hand, the graph model is expressive enough to characterize important OO properties such as aliasing and reachability.

As for graph models of OO programs, there is some work that proposes an OO execution semantics [8,3]. However, a graph in their model is a mixture of class structure, object configuration together with commands to be executed and thus difficult to comprehend. It is not clear either how assertions can be formulated and reasoned about. The notion of trace in our graph model comes from [7], a trace model for pointers and objects. But the main concern of their work is to maintain the aliasing information.

There is some work on Hoare logic for reasoning about OO programs. In particular, Pierik and De Boer's logic [14] based on term substitution is close to our work. But different from our approach, they calculate the weakest precondition for both assignment and object creation, and a complicated form of substitution for dynamic allocation of objects is needed. Von Oheimb and Nipkow [17] present a machine-checked Hoare logic for a Java-like language in Isabelle. They use a semantic representation of assertions to manipulate the program state explicitly instead of syntactic term substitution. Similarly, Poetzsch-Heffter and Müller [15] use an explicit object store in their logic and present axioms for manipulating the store. Recently, separation logic [16] has been applied for reasoning about OO languages [13]. By heap separation, aliasing can be handled in a natural way and modularity of reasoning is achieved. But meanwhile, users should be careful of information on separation of heaps for writing correct assertions.

Future work includes the proof of the completeness of the logic, i.e., every valid specification can be proved by the system. In fact, we are quite confident that it is complete, because the logic rules are indeed provided to deal with every kind of program constructs and the only difficult case is recursive method invocation. Besides the development of the theory, it is also important to apply the logic to a more substantial case study and further to investigate tool support for application of automated techniques of verification and analysis of OO programs.

References

1. Apt, K., de Bakker, J.: Semantics and proof theory of pascal procedures. In: Salomaa, A., Steinby, M. (eds.) ICALP 1977. LNCS, vol. 52, pp. 30–44. Springer, Heidelberg (1977)
2. 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 (2009)
3. Corradini, A., Dotti, F.L., Foss, L., Ribeiro, L.: Translating java code to graph transformation systems. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 383–398. Springer, Heidelberg (2004)
4. He, J., Liu, Z., Li, X.: rCOS: A refinement calculus of object systems. Theor. Comput. Sci. 365(1-2), 109–142 (2006)
5. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
6. Hoare, C.A.R.: Procedures and parameters: An axiomatic approach. In: Symposium on Semantics of Algorithmic Languages. Lecture Notes in Mathematics, vol. 188, pp. 102–116. Springer (1971)
7. Hoare, C.A.R., He, J.: A trace model for pointers and objects. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 1–17. Springer, Heidelberg (1999)
8. Kastenberg, H., Kleppe, A., Rensink, A.: Defining object-oriented execution semantics using graph transformations. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 186–201. Springer, Heidelberg (2006)
9. Ke, W., Liu, Z., Wang, S., Zhao, L.: A graph-based operational semantics of OO programs. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 347–366. Springer, Heidelberg (2009)
10. Ke, W., Liu, Z., Wang, S., Zhao, L.: Graph-based type system, operational semantics and implementation of an object-oriented programming language. Technical Report 410, UNU-IIST, P.O. Box 3058, Macau (2009), `http://www.iist.unu.edu/www/docs/techreports/reports/report410.pdf`
11. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. 16(3), 872–923 (1994)
12. Liu, Z., Morisset, C., Stolz, V.: rCOS: theory and tools for component-based model driven development. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 62–80. Springer, Heidelberg (2010)
13. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: POPL 2005, pp. 247–258. ACM, New York (2005)
14. Pierik, C., de Boer, F.S.: A syntax-directed Hoare logic for object-oriented programming concepts. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 64–78. Springer, Heidelberg (2003)

15. Poetzsch-Heffter, A., Müller, P.: A programming logic for sequential Java. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 162–176. Springer, Heidelberg (1999)
16. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS 2002, pp. 55–74. IEEE Computer Society (2002)
17. von Oheimb, D., Nipkow, T.: Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 89–105. Springer, Heidelberg (2002)
18. Zhao, L., Liu, X., Liu, Z., Qiu, Z.: Graph transformations for object-oriented refinement. Formal Aspects of Computing 21(1-2), 103–131 (2009)
19. Zhao, L., Wang, S., Liu, Z.: Graph-based object-oriented Hoare logic. Technical Report 458, UNU-IIST, P.O. Box 3058, Macau (2012), `http://iist.unu.edu/sites/iist.unu.edu/files/biblio/report458.pdf`

Towards a Modeling Language for Cyber-Physical Systems

Longfei Zhu[1], Yongxin Zhao[2], Huibiao Zhu[1], and Qiwen Xu[3]

[1] East China Normal University, Shanghai, P.R. China
[2] National University of Singapore, Singapore
[3] University of Macau, Macau SAR, P.R. China
{lfzhu,hbzhu}@sei.ecnu.edu.cn, zhaoyx@comp.nus.edu.sg, qwxu@umac.mo

Abstract. A cyber-physical system (CPS) is an interactive system of continuous plants and real-time controller programs. These systems usually feature a tight relationship between the physical and computational components and exhibit true concurrency with respect to time. These communication and concurrency issues have been well investigated in event based synchronous languages but only for discrete systems. In this paper, we investigate the distinct features of CPS and propose an imperative-style language framework for the programming of CPS. To characterize the semantics of the language, a set of algebraic laws are provided, which can be used to reduce arbitrary program into normal form. The programs in the normal form exhibit clear time-consuming and instantaneous behaviors. Moreover, the algebraic laws can be used in the transformation from the high level hybrid program specification to low level controller programs interacting with the physical plants. We will investigate this part in the follow-up work.

1 Introduction

A cyber-physical system (CPS) is an interactive system of continuous plants and real-time controller programs. The embedded controllers monitor and control the evolution of physical plants and make the whole system behave correctly and safely. Such systems are pervasively used in areas such as aerospace, automotive, chemical processes, energy, health-care, etc. It is drawing increasing attention because of its complex but wide applications and high safety requirements in the applications.

For the modeling and verification of cyber-physical systems, varieties of formal models have been investigated in [1–12] among others. Our intention is to explore a provably correct way of development from specification to implementation. Crucial in our approach are a set of algebraic laws that could transform the arbitrary programs to a representation which is either easy to prove the correctness or close to the implementation. In this paper we will emphasize on the algebraic laws that could simplify the program texts and expose the meanings of the programs.

The great merit of algebra is that it makes no distinctions between the specification and implementation [13, 14]. Thus we can represent and reason about the

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 394–411, 2013.

specification and the implementation in one framework. Moreover, the algebra is suited for symbolic calculation and could be used as the basis of reasoning. Algebraic proof via term rewriting is one of the most promising way that computers could assist in the process of reasoning. In practice, the algebraic approach has been successfully applied in the ProCoS project (Provably Correct Systems) on the verification of compiler [15]. In [16], a large number of elegant algebraic laws in OCCAM were provided to characterize the language's semantics. Within the OCCAM language framework, a provably correct compilation method from a high-level OCCAM program to a netlist of Field Programmable Gate Array components via algebraic laws was constructed [17]. The hardware/software partitioning problem could be tackled via the algebra approach as well [18].

To reuse most of the memorable laws in OCCAM and the algebraic transformation strategies, we propose a conservative extension to the language with changes on the communication mechanism due to the characteristics of CPS. Similar to OCCAM, the event is still the only way to synchronize the behaviors of parallel processes. An input event (or event guard) can be triggered only when the corresponding output event becomes ready. However, the output in event based communication paradigm presented in this paper is non-blocking, i.e., the event can be outputted regardless of the environment. Moreover, the output is broadcast among the components, which means the communications could happen among both parallel components and sequential components.

Another main communication paradigm of parallel programming is based on shared variables [19, 20]. The composition behaviors are arbitrary interleavings of parallel processes and the shared variables could be updated several times as all the other discrete variables even at the same time instant. In the event based communication paradigm, the evaluation of the communication events which reflect the change of physical world is related to the time, i.e., the value of each event can be updated at most once at the same time instant. Comparing with shared variable based concurrency, the event based communication mechanism can reduce the non-determinism to some extent but bring in the complexity of iterative triggering of event guards.

The communication issues have been investigated in event based synchronous languages but only for discrete systems [21–24]. To deal with the physical components in CPS, we introduce the continuous statements to the language. More algebraic laws are presented to characterize the semantics of language. The algebraic laws can be used to reduce arbitrary program into normal form. The transformation happens to remove all the iterative triggering of event guards in one instantaneous lock-step. The programs in the normal form exhibit clear time-consuming and instantaneous behaviors.

The remaining part is organized as follows. Section 2 introduces the features of our model and a few assumptions. A set of algebraic laws are presented in Section 3. Section 4 specifies the normal form of the language. Section 5 concludes the paper and outlines some future work.

2 The Cyber-Physical System Domain

Before introducing our approach, we would like first illustrate the features of CPS in our model. From the architecture point of view, CPS can be classified into sensors, actuators, digital components and physical components. It exhibits both continuous and discrete behaviors which interact tightly. The event is the mechanism that the components interact with the others. For example, on the one hand, the change of continuous variables can be viewed as an event, which can be detected by the sensor and thereby influence the execution of digital parts. On the other hand, the event can be explicitly emitted by the digital programs and consequently influences the continuous evolution.

2.1 The Variables

We would like firstly classify all the variables involved in cyber-physical systems. Fig. 1 illustrates different categories of variables. Note that these categories are disjoint.

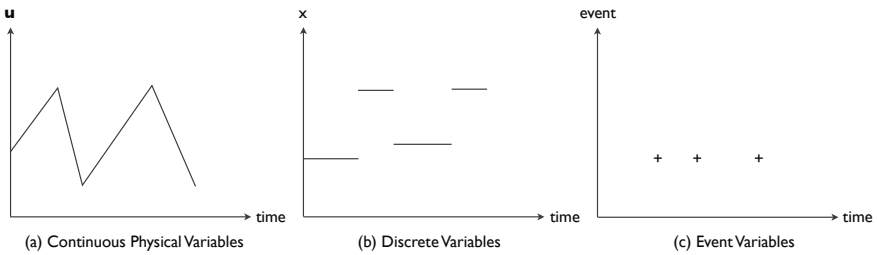


Fig. 1. The evaluation for different categories of variables

- Continuous Physical Variables. Variables in this category are dominated by natural laws and vary with respect to time. The variability of these variables can be described via differential equations. We can only change the value via modifying the algebraic variables involved in differential-algebraic equations or differential equations itself, i.e., the mode of variability. We use **u**, **v**, **x** to denote the continuous variables. Examples in this category are those physical variables such as the height of the bouncing ball, the level of water tank, etc.
- Discrete Variables. Informally, the discrete variables are program variables which are piecewise constant and can be changed at each event instant. They are not shared between controllers and plants.
- Event Variables. The event could be either present or absent. The event is present when the event is implicitly generated by the changes of continuous variables or explicitly emitted by the controller programs at some time instant. The present state can be immediately seen by the neighbors (processes running in parallel) or the successors. The state of the event is cleared automatically (reset to absent) if the time advances.

2.2 The Communication Mechanism

Similar to Communicating Sequential Processes (CSP) [25], the input is synchronous, i.e., an input guard is waiting to be triggered until the corresponding output event becomes ready. The difference is that in our model the output is asynchronous, i.e., output can be executed without the need for the ready input. The synchronous input and asynchronous output reflect the characteristics of CPS, in which some components are driven by the events from the environment but the event can be outputted regardless of the environment. For example, the sensor monitors the evolution of water level and is triggered if the water rises to a dangerous level. But the event representing such emergent situation can always be outputted no matter whether the sensor exists or not.

The second feature is that the output is broadcast among the components, which means that communications can take place among both parallel components and sequential components. An input guard can be fired by the event outputted by the parallel components or its sequential predecessor.

The output events will propagate at each time instant. That means the event will always be present as long as time does not elapse. It is different from the view adopted by Hardware Description Language Verilog [26, 27]. In Verilog, the so-called microstep (or micro-time) is introduced to deal with the execution at the same time instant. Only one process can output a event which could influence the execution at each microstep. Once the event triggers the input guard and the microstep advances, the event will be refreshed and cannot trigger the guards in the successors any longer. For example, if one input guard is followed by the same input guard, the second input guard will not be triggered. But in our CPS model, the present state of event will propagate in one lock-step and the event guards could be iteratively triggered if the time does not advance. In other words, once the input guard can be triggered it can always be triggered provided that the time does not elapse.

2.3 The Concurrency

The cyber-physical system is a true concurrent model and concurrency should be ensured at every time instant. During continuous evolution, the differential equations that describe the relations between the variables should be satisfied concurrently. As for the discrete part, the transitions happen instantaneously (comparing to the macro time, the execution time of digital computation can be abstracted away, but the causality relation is reserved). The instantaneous discrete transitions are executed in a lock-step. Once the instantaneous transition starts, all the discrete transitions are executed till the system reaches to a stable state.

The whole system should be consistent with respect to the time, which means at every time instant, the observation should be consistent. During continuous evolution, the principle is easily fulfilled if all the differential equations are satisfied concurrently. As for the discrete part, the subsequent execution in one lock-step in the computation could not contradict the observation in the same

lock-step. And the effect of instantaneous statements will be observed immediately by the neighbors and successors.

3 The Language and Algebraic Laws

To deal with the features of CPS, continuous statement and event emission statement are introduced to the structure of the language. Let us firstly introduce the syntax of the language.

$$\begin{array}{rcl} EQ & ::= & \text{IDLE} \mid F(\dot{\mathbf{u}}, \mathbf{u}) = 0 \mid (EQ|EQ) \\ AP & ::= & II \mid x := v \mid !e \mid EQ \text{ UNTIL } g \\ P & ::= & A \mid P; P \mid P \parallel P \mid \text{WHEN}(G) \mid \text{IF}(C) \mid P^* \\ C & ::= & b \to P \mid C \,\Box\, C \\ G & ::= & g \& P \mid G \,\Box\, G \end{array}$$

Differential equation $F(\dot{\mathbf{u}}, \mathbf{u}) = 0$ is added to express the continuous evolution of physical plant. IDLE represents the never-terminating continuous evolution in which the continuous variables remain unchanged. $EQ|EQ$ represents a set of differential equations.

Informally, II does nothing. $x := v$ is an instantaneous assignment statement. $!e$ is the explicit emission of event e. Continuous statement EQ UNTIL g behaves as EQ when g is not triggered. $P; Q$ is sequential composition and $P \parallel Q$ defines parallel composition. WHEN(G) represents the input event guard statement (await statement). G is the guard construct. The corresponding branch P in G is allowed to execute when the guard g is triggered. If more than one event guard are satisfied at the same time, the branch will be chosen nondeterministically. IF(C) represents the conditional statement. The continuous variables cannot be involved in the Boolean conditions. The repetition statement P^* represents the finite iteration of P. Systems are assumed not to have infinite behaviors. In other words, an infinite sequence of discrete events over a finite time interval (Zeno behavior) is not allowed.

The precedence rules of the operators are listed as follows.

$:=$ (!), $\to$ (&), $\Box$, ;, $\parallel$ (from the tightest to the loosest)

Here in this paper, we adopt the following name conventions. We introduce $(x, y.., z)$ to represent a list of variables.

v, f	expressions
x, y, z	variables
a, b, c	Boolean expressions
g, h, k	event guards
e, s, t	events
E, S, T	event sets
P, Q, R	programs

Note that the algebraic laws in this paper are not exhaustive. We concentrate on the laws that are needed in the transformation from arbitrary programs into

the normal form. And also we do not list the laws of assignment repetitively in this paper as the assignment of discrete variables in this paper share the same laws presented in [16].

3.1 The Guards

The event is used to denote the interaction or communication mechanism of CPS. It may reflect the change of a variable's value, the satisfiability of a desired condition, or the reaching of a desired point of time, etc. The explicitly emitted events can be used to model the broadcast message passing. Different types of events share the following characteristics: (i) each event exhibits one value at the same time instant; (ii) the value of event is cleared when the time advances. The event guards are constructed as follows.

$$g ::= \epsilon \mid \emptyset \mid e \mid g + g \mid g \cdot g \mid b \bullet g$$

Event guard ϵ can be triggered by any event. No event can trigger the guard $\emptyset$. e represents the atomic event guard. The guard composition $g_1 \cdot g_2$ can be fired by the events which triggers both the operands. The event firing either g_1 or g_2 can trigger guard $g_1 + g_2$. The operator $\cdot$ has higher priority than $+$. $b \bullet g$ is triggered when g is triggered and the Boolean expression b is $true$. Both continuous and discrete variables could be involved in the Boolean expressions. The operator $\bullet$ has the highest priority.

There are some laws on the guard operators. Multiplication $\cdot$ is idempotent, commutative, associative and distributes over addition $+$. It has $\emptyset$ as its zero and ϵ as its unit. Addition $+$ is idempotent, commutative, associative. It has $\emptyset$ as its unit and ϵ as its zero. Additionally, if the Boolean expressions do not contain continuous variables, the following laws hold.

$(\cdot - 1)$ $g \cdot g = g$
$(\cdot - 2)$ $g_1 \cdot g_2 = g_2 \cdot g_1$
$(\cdot - 3)$ $g_1 \cdot (g_2 \cdot g_3) = (g_1 \cdot g_2) \cdot g_3$
$(\cdot - 4)$ $\emptyset \cdot g = \emptyset$
$(\cdot - 5)$ $\epsilon \cdot g = g$
$(\cdot - 6)$ $g_1 \cdot (g_1 + g_2) = g_1$
$(\cdot - 7)$ $g \cdot (h_1 + h_2) = g \cdot h_1 + g \cdot h_2$
$(+ - 1)$ $g + g = g$
$(+ - 2)$ $g_1 + g_2 = g_2 + g_1$
$(+ - 3)$ $g_1 + (g_2 + g_3) = (g_1 + g_2) + g_3$
$(+ - 4)$ $\epsilon + e = \epsilon$
$(+ - 5)$ $e + (e \cdot f) = e$
$(\bullet - 1)$ $true \bullet g = g$
$(\bullet - 2)$ $false \bullet g = \emptyset$
$(\bullet - 3)$ $b_1 \bullet g + b_2 \bullet g = (b_1 \vee b_2) \bullet g$
$(\bullet - 4)$ $b_1 \bullet g_1 \cdot b_2 \bullet g_2 = (b_1 \wedge b_2) \bullet (g_1 \cdot g_2)$

The partial relation $\leq$ reflects how the guard can be triggered: if the smaller guard can be triggered, the larger guard can also be triggered.

Definition 1. $g_1 \leq g_2 =_{df} g_1 \cdot g_2 = g_1$.

Lemma 1. *(1)* $\emptyset \leq g$; $g \leq \epsilon$;
(2) $g_1 \cdot g_2 \leq g_1 \leq g_1 + g_2$;
(3) If $g_1 \leq g_2$, *then* $g_1 \cdot g \leq g_2 \cdot g$

3.2 Laws of Instantaneous Statements

The statements can be further divided into instantaneous ones and time-consuming ones. Here in this subsection, we mainly focus on the laws of instantaneous statements.

Definition 2. P is *instantaneous* ($P \in I$) if P is in the following forms.

- $x := v$, $!e$, or II.
- $\mathrm{IF}(\Box_{i \in I} b_i \to P_i)$ if P_i is instantaneous.
- $P \| Q$, or $P; Q$ if both the operands are instantaneous.

The order of assignment and emission is not relevant.

(instan - 1) $x := v; !e \;=\; !e; x := v \;=\; !e \| x := v$

For the composition of emission statements, the order is irrelevant.

(instan - 2) $!e; !s \;=\; !s; !e \;=\; !e \| !s \;=\; !\{e, s\}$

The emission of event will not change the variables in any c_i.

(instan - 3) $!e; \mathrm{IF}(\Box_{i \in I}\{c_i \to P_i\}) \;=\; \mathrm{IF}(\Box_{i \in I}\{c_i \to (!e; P_i)\})$
provided that $\bigvee_{i \in I} c_i = true$

The assignment can be distributed over the IF construct.

(instan - 4) $x := v; \mathrm{IF}(\Box_{i \in I}\{c_i \to P_i\}) \;=\; \mathrm{IF}(\Box_{i \in I}\{c_i[v/x] \to (x := v; P_i)\})$
provided that $\bigvee_{i \in I} c_i = true$

The following distribution laws are still valid.

(instan - 5) $\mathrm{IF}(\Box_{i \in I}\{b_i \to P_i\}); Q \;=\; \mathrm{IF}(\Box_{i \in I}\{b_i \to (P_i; Q)\})$
provided that $\bigvee_{i \in I} b_i = true$

(instan - 6) $\mathrm{IF}(\Box_{i \in I}\{b_i \to P_i\}) \| Q \;=\; \mathrm{IF}(\Box_{i \in I}\{b_i \to (P_i \| Q)\})$
provided that $\bigvee_{i \in I} b_i = true$

The nested IF statements can be eliminated via the following law.

(instan - 7) $\mathrm{IF}(c \to \mathrm{IF}(\Box_{i \in I}\{b_i \to P_i\})) \;=\; \mathrm{IF}(\Box_{i \in I}\{c \wedge b_i \to P_i\})$

If none of the Boolean conditions are evaluated to *true*, the process behaves as II.

(instan - 8) $\mathrm{IF}(\Box_{i\in I}\{b_i \to P_i\}) \;=\; \mathrm{IF}(\Box_{i\in I}\{b_i \to P_i\} \,\Box\, \neg \bigvee_{i\in I} b_i \to \mathit{II})$

The parallel processes are disjoint on the discrete variables. So we can convert the parallel of assignments into a multiple assignment.

(instan - 9) $x := v \| y := f \;=\; (x, y) := (v, f)$

3.3 Laws of WHEN

The guard construct is associative, commutative, and idempotent.

($\Box$ - 1) $(G_1 \,\Box\, G_2) \,\Box\, G_3 \;=\; G_1 \,\Box\, (G_2 \,\Box\, G_3)$

($\Box$ - 2) $G_1 \;\Box\; G_2 \;=\; G_2 \;\Box\, G_1$

($\Box$ - 3) $G \,\Box\, G \;=\; G$

The component guarded by $\emptyset$ will never be triggered.

(WHEN - 1) $\mathrm{WHEN}(\emptyset \& P \,\Box\, G) \;=\; \mathrm{WHEN}(G)$

The await statement with $\emptyset$ guard behaves as IDLE. Here, αP stands for the alphabet of the process P.

(WHEN - 2) $\mathrm{WHEN}(\emptyset \& P) \;=\; \mathrm{IDLE}_{\alpha P}$

The guarded choice with the same guarded component in an await statement can be combined.

(WHEN - 3) $\mathrm{WHEN}(g \& P \,\Box\, h \& P \,\Box\, G) \;=\; \mathrm{WHEN}((g + h) \& P \,\Box\, G)$

The WHEN can be distributed over the sequential operator.

(WHEN - 4) $\mathrm{WHEN}(\Box_{i\in I}\{g_i \& P_i\}); Q \;=\; \mathrm{WHEN}(\Box_{i\in I}\{g_i \& (P_i; Q)\})$

The following two laws are crucial since they depict the propagation of events and show the elimination of iterative triggering of event guards.

The nested await statements can be converted into single await statement with Boolean event guards. If the instantaneous statement cannot trigger the next await statement, a continuous statement is added explicitly. Otherwise, the guard in the next await statement will be absorbed.

(WHEN - 5) Let A represent an instantaneous program without parallel constructs and $Q = \text{WHEN}(\Box_{i\in I}\{h_i \& Q_i\})$, then

$$\text{WHEN}(g\&(A;Q)\,\Box\,G) = \text{WHEN}\begin{pmatrix} \neg(g \geq A(h)) \bullet g\&(A;\text{IDLE UNTIL } h;Q) \\ \Box\,\Box_{i\in I}\{g \geq A(h_i) \bullet g\&(A;Q_i)\} \\ \Box\,G \end{pmatrix},$$

where $h = \sum_{i\in I} h_i$ and $A(h)$ can be defined inductively on the structure of A as follows.

(1) $(I\!I)(h) = h$ (2) $(!e)(h) = h[\epsilon/e]$

(3) $(x := v)(h) = h[v/x]$ (4) $(A_1; A_2)(h) = A_1(A_2(h))$

(5) $(\text{IF}(\Box_{i\in I}\{b_i \to A_i\}))(h) = \sum_{i\in I} b_i \bullet A_i(h)$

The continuous statement nested in the await statement can be transformed in the similar way. $A(h)$ is defined in the same way as shown in Law WHEN - 5.

(WHEN - 6) Let A represent an instantaneous program without parallel constructs,

$$\text{WHEN}(g\&(A;EQ \text{ UNTIL } h;R)\,\Box\,G) = \text{WHEN}\begin{pmatrix} \neg(g \geq A(h) \bullet g\&(A;EQ \text{ UNTIL } h;R) \\ \Box\, g \geq A(h) \bullet g\&(A;R) \\ \Box\,G \end{pmatrix}.$$

The parallel of two await statements can be expanded into an await statement. For the resulted await statement, there are three cases. The first and the second are composed of a set of event guard components from one parallel branch. The third type of guarded choice describes the common event guard of the two parallel parts, i.e., the compound event guard of the parallel processes.

(WHEN - 7) Let $P = \text{WHEN}(\Box_{i\in I}\{g_i \& P_i\})$ and $Q = \text{WHEN}(\Box_{j\in J}\{h_j \& Q_j\})$,

$$P\|Q = \text{WHEN}\begin{pmatrix} \Box_{i\in I}\,\{g_i\&(P_i\|Q)\} \\ \Box\,\Box_{j\in J}\{h_j\&(Q_j\|P)\} \\ \Box\,\Box_{i\in I, j\in J}\{g_i \cdot h_j\&(P_i\|Q_j)\} \end{pmatrix}.$$

The IF construct can be transformed into an await statement.

(WHEN - 8) $\text{IF}(\Box_{i\in I}\{b_i \to P_i\}) = \text{WHEN}(\Box_{i\in I}\{b_i \bullet \epsilon\ \&P_i\})$

3.4 Laws of UNTIL

If the same equations are put together, it is equal to one equation.

(UNTIL - 1) $(EQ|EQ)$ UNTIL $g \;=\; EQ$ UNTIL g

If one element of the simultaneous equation set is IDLE, it can be dropped. The alphabet becomes the union of the two parts.

(UNTIL - 2) $(EQ_\alpha|\text{IDLE}_\beta)$ UNTIL $g \;=\; EQ_{\alpha\cup\beta}$ UNTIL g

If the guard in the continuous statement is triggered at the beginning, the statement behaves as II.

(UNTIL - 3) EQ UNTIL $\epsilon \;=\; II$

Continuous statement with event guard $\emptyset$ is non-terminating. Any statement after a non-terminating one will never been executed.

(UNTIL - 4) EQ UNTIL $\emptyset; Q \;=\; EQ$ UNTIL $\emptyset$

If the continuous statement is terminated, the event that terminates the continuous statement propagates provided that the time does not elapse.

(UNTIL - 5) EQ UNTIL $g; \text{WHEN}(g\&P) \;=\; EQ$ UNTIL $g; P$

We can merge the continuous statements with the same differential equation.

(UNTIL - 6) EQ UNTIL $(g+h); EQ$ UNTIL $g \;=\; EQ$ UNTIL g

The parallel of two continuous statements can be transformed into a sequential one.

(UNTIL - 7) Let $P_1 \;=\; EQ_1$ UNTIL $g_1; R_1$
and $P_2 \;=\; EQ_2$ UNTIL $g_2; R_2$
then $P_1 \| P_2 \;=\; (EQ_1|EQ_2)$ UNTIL (g_1+g_2);
$\text{WHEN}(g_1\&(R_1\|P_2) \,\Box\, g_2\&(R_2\|P_1) \,\Box\, g_1 \cdot g_2\&(R_1\|R_2))$.

We can eliminate the parallel operator via the following law if it is combined by the continuous statement and await statement.

(UNTIL - 8) Let $P \;=\; EQ$ UNTIL $h; R$
and $Q \;=\; \text{WHEN}(\Box_{j\in J} g_j\&Q_j)$
then $P\|Q \;=\; EQ$ UNTIL $(\sum_{j\in J} g_j + h)$;
$\text{WHEN}(\Box_{j\in J} g_j\&(P\|Q_j) \,\Box\, h\&(R\|Q) \,\Box\, \Box_{j\in J} h \cdot g_j\&(R\|Q_j))$.

Example 1. In the following example, we show how to eliminate the iterative triggering of event guards in the await statements via the above algebraic laws.

$$\text{WHEN}(s\&!e);\text{WHEN}(s\cdot e\&!t)$$

(by Law WHEN - 4)

$$= \text{WHEN}(s\&!e;\text{WHEN}(s\cdot e\&!t))$$

(by Law WHEN - 5)

$$= \text{WHEN}\left(\begin{array}{l}\neg(s\geq(!e)(s\cdot e))\bullet s\&\left(!e;\text{IDLE UNTIL}\,(s\cdot e);\text{WHEN}(s\cdot e\&!t)\right)\\ \Box\,(s\geq(!e)(s\cdot e))\bullet s\&(!e;!t)\end{array}\right)$$

(by the definition of $\geq$)

$$= \text{WHEN}\left(\begin{array}{l}false\bullet s\&\left(!e;\text{IDLE UNTIL}\,(s\cdot e);\text{WHEN}(s\cdot e\&!t)\right)\\ \Box\,true\bullet s\&(!e;!t)\end{array}\right)$$

(by $\bullet$ - 1, $\bullet$ - 2)

$$= \text{WHEN}(\emptyset\&(!e;\text{WHEN}(s\cdot e\&!t))\,\Box\, s\&(!e;!t))$$

(by Law WHEN - 1)

$$= \text{WHEN}(s\&(!e;!t))$$

4 The Normal Form

To illustrate the power of the algebraic laws, we investigate the normal form of our programs and prove that all the programs can be converted into a normal form. In the transformation process, the iterative triggering of event guards is eliminated. Thus the normal form exhibits the clear alternate instantaneous and time-consuming behaviors, i.e., each instantaneous execution is followed by an explicitly continuous evolution and the continuous evolution is followed by instantaneous execution.

In this section, we firstly present the normal form for the instantaneous statements. After that, several derived laws are introduced to facilitate the transformation. Then we give the normal form of the programs. The proof also demonstrates the process to transform the program text into a normal form.

4.1 Conditional Normal Form for Instantaneous Statements

Every finite instantaneous program of our language can be reduced into a simple conditional normal form. Here we extend the event emission statement and use $!E$ to denote the emission of a set of event. In conditional normal form a program looks like

$$\text{IF}(\Box_{i\in I}\{b_i \to P_i\}),$$

where $\bigvee_{i\in I} b_i = true$ and P_i is $x := v_i \| !E_i$. The sequential operator is eliminated.

Theorem 1. *All the instantaneous statements can be reduced to conditional normal form.*

Proof. Our first step is to show that all primitives can be reduced to conditional normal form. If we can further prove that normal form is closed under the combinators, it is sufficient to know that all the instantaneous statements can be reduced to conditional normal form. The proof proceeds as follows.
(1) II.

$$\mathit{II} \;=\; \mathrm{IF}(\mathit{true} \to x := x \| !\emptyset).$$

(2) Emission.

$$!e \;=\; \mathrm{IF}(\mathit{true} \to x := x \| !\{e\}).$$

(3) Assignment.

$$x := v \;=\; \mathrm{IF}(\mathit{true} \to x := v \| !\emptyset).$$

(4) Conditional Statement.

$$\mathrm{IF}(\Box_{i\in I}\{b_i \to \mathrm{IF}(\Box_{j\in J}\{c_j^i \to (x := v_j \| !E_j)\})\})$$

(by Law instan - 7)

$$= \mathrm{IF}(\Box_{i\in I, j\in J}\{b_i \wedge c_j^i \to (x := v_j \| !E_j)\}).$$

(5) Sequential Composition.

$$\mathrm{IF}(\Box_{i\in I}\{b_i \to (x := v_i \| !E_i)\}); \mathrm{IF}(\Box_{j\in J}\{c_j \to (x := f_j \| !S_j)\})$$

(by Law instan - 5)

$$= \mathrm{IF}(\Box_{i\in I}\{b_i \to ((x := v_i \| !E_i); \mathrm{IF}(\Box_{j\in J}\{c_j \to (x := f_j \| !S_j)\}))\})$$

(by Law instan - 1)

$$= \mathrm{IF}(\Box_{i\in I}\{b_i \to (x := v_i; !E_i; \mathrm{IF}(\Box_{j\in J}\{c_j \to (x := f_j \| !S_j)\}))\})$$

(by Law instan - 3, instan - 4, instan - 7)

$$= \mathrm{IF}(\Box_{i\in I, j\in J}\{b_i \wedge c_j[v_i/x] \to ((x := v_i; x := f_j) || !E_i \cup S_j)\}).$$

(6) Parallel Composition.

$$\mathrm{IF}(\Box_{i\in I}\{b_i \to (x := v_i \| !E_i)\}) \| \mathrm{IF}(\Box_{j\in J}\{c_j \to (y := f_j \| !S_j)\})$$

(by Law instan - 6)

$$= \mathrm{IF}(\Box_{i\in I}\{b_i \to ((x := v_i \| !E_i) \| \mathrm{IF}(\Box_{j\in J}\{c_j \to (y := f_j \| !S_j)\}))\})$$

(by comm of $\|$, Law instan - 6)

$$= \mathrm{IF}(\Box_{i\in I}\{b_i \to \mathrm{IF}(\Box_{j\in J}\{c_j \to ((x := v_i \| !E_i) \| (y := f_j \| !S_j))\})\})$$

(by Law instan - 7, instan - 9, instan - 2)

$$= \mathrm{IF}(\Box_{i\in I, j\in J}\{b_i \wedge c_j \to ((x, y := v_i, f_j) || !E_i \cup S_j)\}).$$

□

4.2 Additional Derived Laws

Here we firstly introduce three parallel expansion laws. They depict how to transform the parallel construct into sequential one. Note that the Boolean conditions in the following derived parallel expansion laws mentioned below partition true.

If the first statements of the parallel processes are both instantaneous, the effect of execution can be merged.

(der - 1) Let $P = \mathrm{IF}(\Box_{i\in I}\{b_i \to ((x := v_i \| !E_i); P_i)\})$
and $Q = \mathrm{IF}(\Box_{j\in J}\{c_j \to ((y := f_j \| !S_j); Q_i)\})$,
then $P \| Q = \mathrm{IF}(\Box_{i\in I, j\in J}\{b_i \wedge c_j \to (((x,y) := (v_i, f_j) \| !E_i \cup S_j); (P_i \| Q_j))\})$.

Proof. See the appendix. □

The instantaneous statements are executed before the continuous ones.

(der - 2) Let $P = \mathrm{IF}(\Box_{i\in I}\{b_i \to x := v_i \| !E_i; P_i\})$
and $Q = EQ$ UNTIL $g; Q'$
then $P \| Q = \mathrm{IF}(\Box_{i\in I}\{b_i \to ((x := v_i \| !E_i); (P_i \| Q))\})$.

Proof. See the appendix. □

If instantaneous statements and await statements are executed in parallel, the instantaneous ones are executed first.

(der - 3) Let $P = \mathrm{IF}(\Box_{i\in I}\{b_i \to x := v_i \| !E_i; P_i\})$
and $Q = \mathrm{WHEN}(\Box_{j\in J}\{g_j \& Q_j\})$,
then $P \| Q = \mathrm{IF}(\Box_{i\in I}\{b_i \to ((x := v_i \| !E_i); (P_i \| Q))\})$.

The following two derived laws are simple variants of Law WHEN - 5, WHEN - 6 (under Law instan - 1 and instan - 2). $h[\epsilon/E]$ represents the substitution of every element of set E in h with ϵ.

(der - 4) Let $Q = \mathrm{WHEN}(\Box_{i\in I}\{h_i \& Q_i\})$ and $h = \sum_{i\in I} h_i$, then

$$\mathrm{WHEN}(g\&((x := v \| !E); Q) \,\Box\, G) = \mathrm{WHEN}\left(\begin{array}{l} \neg(g \geq h[v/x, \epsilon/E]) \bullet g\& \left(\begin{array}{l}(x := v \| !E);\\ \mathrm{IDLE\ UNTIL}\ h;\\ Q\end{array}\right) \\ \Box\Box_{i\in I}\{g \geq h_i[v/x, \epsilon/E] \bullet g\&((x := v \| !E); Q_i)\} \\ \Box\, G \end{array}\right)$$

(der - 5) Let $h = \sum_{i\in I} h_i$, then

$$\mathrm{WHEN}((g\&(x := v \| !E); EQ\ \mathrm{UNTIL}\ h; R) \,\Box\, G) = \mathrm{WHEN}\left(\begin{array}{l} \neg(g \geq h[v/x, \epsilon/E] \bullet g\& \left(\begin{array}{l}(x := v \| !E);\\ EQ\ \mathrm{UNTIL}\ h;\\ R\end{array}\right) \\ \Box\, g \geq h_i[v/x, \epsilon/E] \bullet g\&((x := v \| !E); R) \\ \Box\, G \end{array}\right)$$

4.3 The Normal Form

After introducing the await statements and continuous statements, it is not easy to distinguish the instantaneous statements from the time-consuming ones. If the guard in the await statements is triggered, the guarded component will be executed at current time instant. The execution may emit more events and the combined events may release more event guards. It is the same with the continuous statements with guards. We use the algebraic laws to eliminate the iterative triggering of event guards.

The transformation eliminates iterative triggering of event guards and the result normal form exhibits clear alternate instantaneous behaviors and time-consuming ones. The normal form is

$$\text{WHEN}(\Box_{i\in I}\{g_i \& P_i\}),$$

where P_i is $x := v \| !E$, or $x := v \| !E; EQ$ UNTIL $g; R$ and R is still in the normal form. Note that the await statement in the normal form will not take time. All the time-consuming behaviors are explicitly represented by the continuous statements.

Theorem 2. *The instantaneous statements can be reduced to normal form.*

Proof. From Theorem 1, all the instantaneous statements can be transformed into a conditional normal form. Based on the Law WHEN - 8, we can convert all the conditional statements into the await statements. □

Theorem 3. *The continuous statement can be reduced to normal form.*

Proof. Since $I\!I$ is the unit of any statement, the continuous statements P can be regarded as

$$\text{WHEN}(true \bullet \epsilon \& (x := x \| !\emptyset; P)).$$

□

Theorem 4. *The sequential composition $P; Q$ can be reduced to normal form, if P and Q are in normal form.*

Proof. Without loss of generality, we assume $P = \text{WHEN}(\Box_{i\in I}\{g_i \& P_i\})$. Based on Law WHEN - 4 (the distribution law &-;), then

$$P; Q \;=\; \text{WHEN}(\Box_{i\in I}\{g_i \& (P_i; Q)\})$$

If P_i is $x := v \| !E$, we need to deal with the cases separately. If Q is await statement, by Law der - 4 we can eliminate the nested await. If Q is a continuous statement, Law der - 5 could be applied to eliminate the iterative triggering of event guards.

If P_i is $x := v \| !E; EQ$ UNTIL g, then $P_i; Q \;=\; x := v \| !E; EQ$ UNTIL g; WHEN($g \& Q$) (by Law UNTIL - 5). We can further apply Law WHEN - 5 to eliminate the iterative triggering of event guards. □

Theorem 5. *The parallel composition $P\|Q$ can be reduced into normal form, if P and Q are in the normal form.*

Proof. We need to analyze different situations in terms of the structure of P and Q. By Law WHEN - 7, UNTIL - 7, UNTIL - 8, we can convert the parallel into a sequential one. The sequential statement can be transformed into normal form by Theorem 4. □

Theorem 6. *The repetition statement P^* can be reduced to normal form if P is in the normal form.*

Proof. We firstly show that for all natural numbers n, P^n can be reduced to normal form if P is in normal form. The proposition can be proved by mathematical induction on natural numbers n. Obviously, it holds when n is equal to 0. Assume that P^n can be reduced into normal form. According to Theorem 4 and $P^{n+1} = P^n; P$, P^{n+1} can be reduced to normal form. Thereby P^n holds for all natural numbers. Thus P^* can be reduced to normal form. □

Theorem 7. *All the statements can be reduced to normal form.*

Proof. From Theorem 2, 3, 4, 5, 6. □

5 Conclusion and Future Work

In this paper, we presented an imperative-style language framework for cyber-physical system modeling. A set of algebraic laws were presented, which characterized the semantics of language. The algebraic laws could be used to reduce arbitrary programs to normal form. The iterative triggering of event guards in the programs were removed in the transformation and the result normal form exhibited clear alternate instantaneous behaviors and time-consuming ones. This paper has illustrated one usage of algebraic laws, i.e., simplifying the program texts and transforming all programs to a representation which exhibits clear behaviors and is easy to prove the correctness.

The second usage of the algebraic laws is the decomposition. CPS can be further divided into at least two parallel parts, namely, the physical plant and digital controller. The algebraic laws can be used to decompose the high level specification into several parts running in parallel which is closer to the implementation. The controller decomposed from the high level specification could serve as the specification for further development. To facilitate the decomposition process, more derived laws are needed. We will investigate this part in the future.

The algebraic laws can be used to characterize the semantics of the language but the soundness of the algebraic laws has not been ensured. To enhance the reliability of our approach, we are planing to explore the denotational semantics of the language and prove the correctness of the algebraic laws with respect to the denotational semantics.

Acknowledgments. This work will not be possible without Professor He Jifeng, nor in fact would be for most of the other work of the same authors. The authors are deeply indebted to Professor He, who is the supervisor of all the four, for the guidance, inspiration and vision in all these years.

This work is supported by National Basic Research Program of China (No. 2011CB302904), National High Technology Research and Development Program of China (No. 2011AA010101 and No. 2012AA011205), National Natural Science Foundation of China (No. 61061130541 and No. 61021004 and No. 91118008), Shanghai Knowledge Service Platform Project (No. ZF1213), the Doctoral Program Foundation of Institutions of Higher Education of China (XRZZ2012022) and Macao Science and Technology Development Fund under the EAE project (No.072/2009/A3).

References

1. Jifeng, H.: A classical mind, pp. 171–189. Prentice Hall International (UK) Ltd., Hertfordshire (1994)
2. Henzinger, T.A.: The theory of hybrid automata. In: LICS, pp. 278–292. IEEE Computer Society (1996)
3. Lynch, N.A., Segala, R., Vaandrager, F.W.: Hybrid i/o automata. Inf. Comput. 185(1), 105–157 (2003)
4. Benveniste, A., Bourke, T., Caillaud, B., Pouzet, M.: A hybrid synchronous language with hierarchical automata: static typing and translation to synchronous code. In: Chakraborty, S., Jerraya, A., Baruah, S.K., Fischmeister, S. (eds.) EMSOFT, pp. 137–148. ACM (2011)
5. Bauer, K., Schneider, K.: From synchronous programs to symbolic representations of hybrid systems. In: Johansson, K.H., Yi, W. (eds.) HSCC, pp. 41–50. ACM (2010)
6. Baldamus, M., Stauner, T.: Modifying esterel concepts to model hybrid systems. Electr. Notes Theor. Comput. Sci. 65(5), 35–49 (2002)
7. Fritzson, P.: Modelica - a cyber-physical modeling language and the openmodelica environment. In: IWCMC, pp. 1648–1653. IEEE (2011)
8. Su, W., Abrial, J.-R., Zhu, H.: Complementary methodologies for developing hybrid systems with event-B. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 230–248. Springer, Heidelberg (2012)
9. Abrial, J.-R., Su, W., Zhu, H.: Formalizing hybrid systems with event-B. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 178–193. Springer, Heidelberg (2012)
10. Platzer, A.: Logical Analysis of Hybrid Systems - Proving Theorems for Complex Dynamics. Springer (2010)
11. Fritzson, P.: The Modelica object-oriented equation-based language and its OpenModelica environment with metamodeling, interoperability, and parallel execution. In: Ando, N., Balakirsky, S., Hemker, T., Reggiani, M., von Stryk, O. (eds.) SIMPAR 2010. LNCS, vol. 6472, pp. 5–14. Springer, Heidelberg (2010)
12. Lee, E.A., Seshia, S.A.: Introduction to Embedded Systems - A Cyber-Physical Systems Approach. 1 edn. Lee and Seshia (2010)
13. Hoare, C.A.R., Hayes, I.J., He, J., Morgan, C., Roscoe, A.W., Sanders, J.W., Sørensen, I.H., Spivey, J.M., Sufrin, B.: Laws of programming. Commun. ACM 30(8), 672–686 (1987)

14. Hoare, T., van Staden, S.: In praise of algebra. Formal Asp. Comput. 24(4-6), 423–431 (2012)
15. He, J., et al.: Provably correct systems. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994 and ProCoS 1994. LNCS, vol. 863, pp. 288–335. Springer, Heidelberg (1994)
16. Roscoe, A.W., Hoare, C.A.R.: The laws of occam programming. Theor. Comput. Sci. 60, 177–229 (1988)
17. He, J., Page, I., Bowen, J.P.: Towards a provably correct hardware implementation of occam. In: Milne, G.J., Pierre, L. (eds.) CHARME 1993. LNCS, vol. 683, pp. 214–225. Springer, Heidelberg (1993)
18. Qin, S., He, J., Qiu, Z., Zhang, N.: An algebraic hardware/software partitioning algorithm. J. Comput. Sci. Technol. 17(3), 284–294 (2002)
19. Qiwen, X., He, J.: Laws of parallel programming with shared variables (1994)
20. Zhu, H., Yang, F., He, J., Bowen, J.P., Sanders, J.W., Qin, S.: Linking operational semantics and algebraic semantics for a probabilistic timed shared-variable language. J. Log. Algebr. Program. 81(1), 2–25 (2012)
21. Zhao, Y., He, J.: Towards a signal calculus for event-based synchronous languages. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 1–13. Springer, Heidelberg (2011)
22. Zhao, Y., Zhu, L., Zhu, H., He, J.: A denotational model for instantaneous signal calculus. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 126–140. Springer, Heidelberg (2012)
23. Berry, G.: The foundations of esterel. In: Plotkin, G.D., Stirling, C., Tofte, M. (eds.) Proof, Language, and Interaction, pp. 425–454. The MIT Press (2000)
24. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Guernic, P.L., de Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE 91(1), 64–83 (2003)
25. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Inc., Upper Saddle River (1985)
26. He, J.: An algebraic approach to the VERILOG programming. In: Aichernig, B.K., Maibaum, T. (eds.) Formal Methods at the Crossroads. From Panacea to Foundational Support. LNCS, vol. 2757, pp. 65–80. Springer, Heidelberg (2003)
27. Zhu, H., He, J., Bowen, J.P.: From algebraic semantics to denotational semantics for Verilog. In: ICECCS, pp. 139–151. IEEE Computer Society (2006)

Appendix

(der - 1) Let $P = \mathrm{IF}(\Box_{i\in I}\{b_i \to ((x := v_i)\|!E_i); P_i)\})$
and $Q = \mathrm{IF}(\Box_{j\in J}\{c_j \to ((y := f_j)\|!S_j); Q_i)\})$,
then $P\|Q = \mathrm{IF}(\Box_{i\in I, j\in J}\{b_i \wedge c_j \to (((x,y) := (v_i, f_j)\|!E_i \cup S_j); (P_i\|Q_j))\})$.

Proof. As the Boolean conditions partition true, we can use the instantaneous laws to prove the derived law.

$P \| Q$

(by Law instan - 6)

$= \text{IF}(\Box_{i \in I}\{b_i \rightarrow (((x := v_i) \| !E_i); P_i) \| Q)\})$

(by comm of $\|$, Law instan - 6, instan - 7, instan - 9, instan - 2)

$= \text{IF}(\Box_{i \in I, j \in J}\{b_i \wedge c_j \rightarrow (((x,y) := (v_i, f_j) \| !E_i \cup S_j); (P_i \| Q_j))\}).$

□

(der - 2) Let $P = \text{IF}(\Box_{i \in I}\{b_i \rightarrow x := v_i \| !E_i; P_i\})$
and $Q = EQ$ UNTIL $g; Q'$
then $P \| Q = \text{IF}(\Box_{i \in I}\{b_i \rightarrow ((x := v_i \| !E_i); (P_i \| Q))\})$.

Proof. We firstly convert conditional statement to await statement and then apply the corresponding expansion laws to eliminate the parallel structure. The proof proceeds as follows.

$P \| Q$

(by Law WHEN - 8)

$= \text{WHEN}(\Box_{i \in I}\{b_i \bullet \epsilon \& ((x := v_i \| !E_i); P_i)\}) \| Q$

(by Law UNTIL - 8)

$= EQ$ UNTIL $(g + \sum_{i \in I} b_i \bullet \epsilon);$

$$\text{WHEN} \left(\begin{array}{l} \Box_{i \in I}\, \{b_i \bullet \epsilon \& (((x := v_i \| !E_i); P_i) \| Q)\} \,\Box\, g \& (P \| Q') \\ \Box\, \Box_{i \in I}\{b_i \bullet \epsilon \cdot g \& (((x := v_i \| !E_i); P_i) \| Q')\} \end{array} \right)$$

(by $+$ - ϵ zero, Law UNTIL - 3)

$= \text{WHEN}(\Box_{i \in I}\{b_i \bullet \epsilon \& (((x := v_i \| !E_i); P_i) \| Q)\})$

(by Law WHEN - 8)

$= \text{IF}(\Box_{i \in I}\{b_i \rightarrow ((x := v_i \| !E_i); (P_i \| Q))\}).$

□

Author Index